TMS320C6000 Optimizing C/C++ Compiler v8.3.x

User's Guide

Literature Number: SPRUI04F
JULY 2015 – REVISED APRIL 2023

Table of Contents

List of Figures

List of Tables

This page intentionally left blank.

Read This First

About This Manual

The *TMS320C6000 Optimizing C/C++ Compiler User's Guide* explains how to use the following Texas Instruments Code Generation compiler tools:

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

The TI compiler accepts C and C++ code conforming to the International Organization for Standardization (ISO) standards for these languages. The compiler supports the 1989, 1999, and 2011 versions of the C language and the 2014 version of the C++ language.

This user's guide discusses the characteristics of the TI C/C++ compiler. It assumes that you already know how to write C/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 World\n");
}
```

- In syntax descriptions, instructions, commands, and directives are 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, the leftmost column is reserved for the first character of a label or symbol. If the label or symbol is optional, it is usually not shown. If a label or symbol 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 the leftmost column.

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

- Some directives can have a varying number of parameters. For example, the .byte directive. This syntax is shown as [, ..., *parameter*].
- This document describes support for the C64+, C6740, and C6600 variants of the TMS320C6000™ processor series. The C6200, C6400, C6700, and C6700+ variants are not supported in v8.0 and later versions of the TI Code Generation Tools. If you are using one of these legacy devices, please use v7.4 of the Code Generation Tools and refer to SPRU187 and SPRU186 for documentation.

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

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 1999 C Standard), International Organization for Standardization

ISO/IEC 9899:2011, International Standard - Programming Languages - C (The 2011 C Standard), International Organization for Standardization

ISO/IEC 14882-2014, International Standard - Programming Languages - C++ (The 2014 C++ Standard), International Organization for Standardization

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

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

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

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 C++ Programming Language (second edition), Bjarne 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

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

DWARF Debugging Information Format Version 4, DWARF Debugging Information Format Workgroup, Free Standards Group, 2010 (http://dwarfstd.org)

System V ABI specification (http://www.sco.com/developers/gabi/)

OpenCL™ Specification version 1.2 (https://www.khronos.org/opencl/)

Related Documentation From Texas Instruments

See the following resources for further information about the TI Code Generation Tools:

- Code Composer Studio Documentation Overview
- Texas Instruments E2E Software Tools Forum

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

SPRUI03 ***TMS320C6000 Assembly Language Tools 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 C64+, C6740, and C6600 variants of the TMS320C6000 platform of devices. Refer to SPRU186 when using legacy C6200, C6400, C6700, and C6700+ devices.

SPRAB89 ***The C6000 Embedded Application Binary Interface***. Provides a specification for the ELF-based Embedded Application Binary Interface (EABI) for the TMS320C6000 family of processors from Texas Instruments. The EABI defines the low-level interface between programs, program components, and the execution environment, including the operating system if one is present.

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).

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.

SPRUGH7 ***TMS320C66x CPU and Instruction Set Reference Guide.*** Describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C66x digital signal processors (DSPs) of the TMS320C6000 DSP platform. The C66x DSP generation comprises floating-point devices in the C6000 DSP platform.

SPRUFE8 ***TMS320C674x CPU and Instruction Set Reference Guide.*** Describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C674x digital signal processors (DSPs) of the TMS320C6000 DSP platform. The C674x is a floating-point DSP that combines the TMS320C67x+ DSP and the TMS320C64x+ DSP instruction set architectures into one core.

SPRAAB5 ***The Impact of DWARF on TI Object Files.*** Describes the Texas Instruments extensions to the DWARF specification.

SPRUEX3 ***TI SYS/BIOS Real-time Operating System User's Guide.*** SYS/BIOS gives application developers the ability to develop embedded real-time software. SYS/BIOS is a scalable real-time kernel. It is designed to be used by applications that require real-time scheduling and synchronization or real-time instrumentation. SYS/BIOS provides preemptive multithreading, hardware abstraction, real-time analysis, and configuration tools.

Trademarks

TMS320C6000™ and Code Composer Studio™ are trademarks of Texas Instruments.
OpenCL™ is a trademark of Apple Inc. used by permission by Khronos.
All trademarks are the property of their respective owners.

This page intentionally left blank.

Chapter 1

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 assembler and linker are discussed in detail in the *TMS320C6000 Assembly Language Tools User's Guide*.

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.

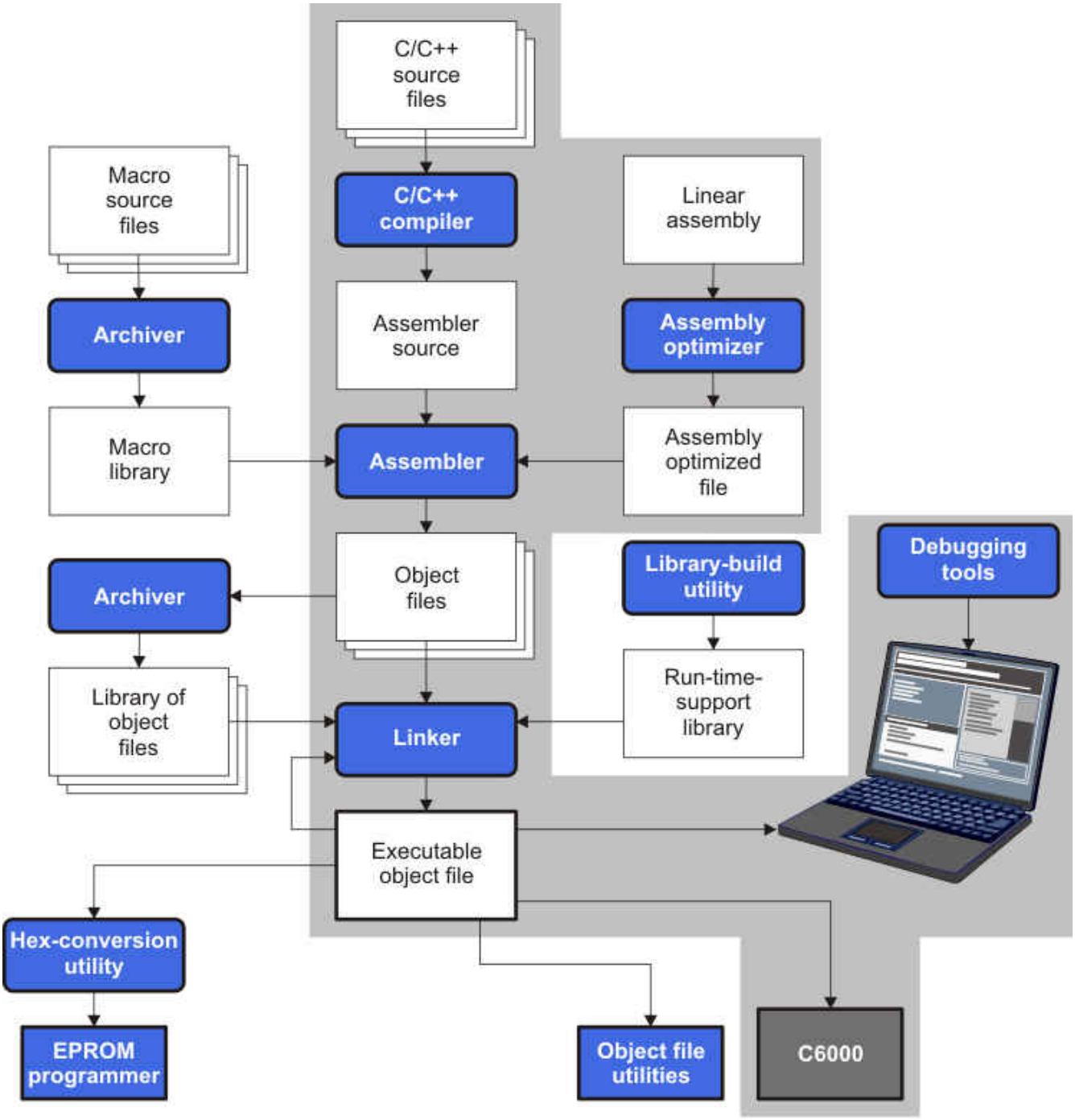


Figure 1-1. TMS320C6000 Software Development Flow

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 5.

- The **compiler** accepts C/C++ source code and produces C6000 assembly language source code. See Chapter 3.
- The **assembler** translates assembly language source files into machine language relocatable object files. See the *TMS320C6000 Assembly Language Tools User's Guide*.
- The **linker** combines relocatable object files into a single absolute executable object file. As it creates the executable file, it performs relocation and resolves external references. The linker accepts relocatable object files and object libraries as input. See Chapter 6 for an overview of the linker. See the *TMS320C6000 Assembly Language Tools User's Guide* for details.
- The **archiver** allows you to collect a group of files into a single archive file, called a *library*. The archiver allows you to modify such libraries by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object files. See the *TMS320C6000 Assembly Language Tools User's Guide*.
- The **run-time-support libraries** contain the standard ISO C and C++ library functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the compiler. See Chapter 9.

  The **library-build utility** automatically builds the run-time-support library if compiler and linker options require a custom version of the library. See Section 9.4. Source code for the standard run-time-support library functions for C and C++ is provided in the lib\src subdirectory of the directory where the compiler is installed.
- The **hex conversion utility** converts an object file into other object formats. You can download the converted file to an EPROM programmer. See the *TMS320C6000 Assembly Language Tools User's Guide*.
- 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 10.
- The **disassembler** decodes object files to show the assembly instructions that they represent. See the *TMS320C6000 Assembly Language Tools User's Guide*.
- The main product of this development process is an executable object file that can be executed on a **TMS320C6000** device. You can use an XDS emulator when refining and correcting your code.

1.2 Compiler Interface

The compiler is a command-line program named cl6x. This program can compile, optimize, assemble, and link programs in a single step. Within Code Composer Studio™, the compiler is run automatically to perform the steps needed to build a project.

For more information about compiling a program, see Section 3.1.

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

1.3 ANSI/ISO Standard

The compiler supports the 1989, 1999, and 2011 versions of the C language and the 2014 version of the C++ language. The C and C++ language features in the compiler are implemented in conformance with the following ISO standards:

- **ISO-standard C:** The C compiler supports the 1989, 1999, and 2011 versions of the C language.
  - **C89.** Compiling with the --c89 option causes the compiler to conform to the ISO/IEC 9899:1990 C standard, which was previously ratified as ANSI X3.159-1989. The names "C89" and "C90" refer to the same programming language. "C89" is used in this document.
  - **C99.** Compiling with the --c99 option causes the compiler to conform to the ISO/IEC 9899:1999 C standard.
  - **C11.** Compiling with the --c11 option causes the compiler to conform to the ISO/IEC 9899:2011 C standard.

The C language is also described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R).

- **ISO-standard C++:** The compiler uses the C++14 version of the C++ standard. Previously, C++03 was used. See the C++ Standard ISO/IEC 14882:2014. For a description of *unsupported* C++ features, see Section 7.2.
- **ISO-standard run-time support:** The compiler tools come with an extensive run-time library. Library functions conform to the ISO C/C++ library standard unless otherwise stated. The library includes functions for standard input and output, string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, and exponential and hyperbolic functions. Functions for signal handling are not included, because these are target-system specific. For more information, see Chapter 9.

See Section 7.13 for command line options to select the C or C++ standard your code uses.

1.4 Output Files

The following type of output file is created by the compiler:

- **ELF object files.** Executable and Linking Format (ELF) enables supporting modern language features like early template instantiation and exporting inline functions. ELF is part of the System V Application Binary Interface (ABI). The ELF format used for C6000 is extended by the C6000 Embedded Application Binary Interface (EABI), which is documented in SPRAB89.

COFF object files are not supported in v8.0 and later versions of the TI Code Generation Tools. If you would like to produce COFF output files, please use v7.4 of the Code Generation Tools and refer to SPRU186 for documentation.

1.5 Utilities

These features are compiler utilities:

- **Library-build utility**

  The library-build utility lets you custom-build object libraries from source for any combination of run-time models. For more information, see Section 9.4.
- **C++ name demangler**

  The C++ name demangler (dem6x) is a debugging aid that translates each mangled name it detects in compiler-generated assembly code, disassembly output, or compiler diagnostic messages to its original name found in the C++ source code. For more information, see Chapter 10.
- **Hex conversion utility**

  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 ELF 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*.

Chapter 2

Getting Started with the Code Generation Tools

This chapter provides an overview of the procedure for creating a Code Composer Studio project that uses the C6000 Code Generation Tools. In addition, it provides an introduction to the command-line for the compiler and linker.

2.1 How Code Composer Studio Projects Use the Compiler

If you use Code Composer Studio (CCS) as your development environment, the compiler and linker options are automatically set for you when you create a project. The project settings you make determine which compiler and linker command line options are used to build the project. Follow these steps to create and build a project in CCS v6.0. The exact steps may vary somewhat in other versions of CCS.

Table 2-1. Steps for Creating a CCS Project

Step	Effects on Use of the Compiler
1. Choose **File > New > CCS Project** from the menus.	
2. In the New CCS Project wizard, first select the **Target**. You can use the drop-down on the left to filter the list of specific targets on the right. The v8.x C6000 Code Generation Tools support C64x+, C6600, and C6740 targets.	Sets the --silicon_version (-mv) compiler option. See Section 3.3.5. In addition, a preprocessor symbol matching the target is defined using the --define compiler option. See Section 3.3.2.
3. In the **Connection** field, select the emulator you will use to connect to the device.	Generates a target configuration file for use when running the project.
4. In the **Project name** field, type a name for the project.	Determines the folder where the project is stored.
5. Expand the **Advanced settings** area.	
6. Make sure the **Compiler version** you want to use is selected.	Sets the --include_path compiler option to the include directory for that version of the Code Generation Tools. See Section 3.5.2.1.
7. By default, C6000 applications are compiled to be little-endian. In the **Device endianness** field, you can choose big-endian if needed.	Sets the --big_endian compiler option if the default is not used. See Section 3.3.4.
8. The linker command file and runtime support library are selected automatically based on your choices in the other fields.	
9. Expand the **Project templates and examples** area.	
10. Select a template for your project. The project templates you can choose from include a completely empty project with no source files, a project containing only main.c, an assembly-only project, and a Hello World example. Other examples that use plug-in software components you have installed are available in the TI Resource Explorer window.	
11. Click **Finish**.	

After you have created a CCS project, you can use the Properties dialog for the project to see how the compiler and linker will be used and modify the command-line options used when compiling and linking. To open this dialog, select the project in the Project Explorer and choose **Project > Properties** from the menus. Expand the category tree to select **Build > C6000 Compiler** and **Build > C6000 Linker**. You can learn more about any command-line options you see in this dialog in Chapter 3.

2.2 Compiling from the Command Line

If you are developing your project outside of an IDE such as Code Composer Studio, you will need to use the command-line interface to the compiler and linker.

The compiler and linker are run using the same executable. This executable is the **cl6x.exe** file, which is located in the **bin** subdirectory of your TI Code Generation Tools installation.

You can use a single command line to both compile your code to create object files and link the object files to create an executable. All the command-line options that occur before the --run_linker (or -z for short) option apply to the compiler. All the command-line options that occur after the --run_linker (-z) option apply to the linker. In the following command-line, the -mv6740, --c99, --opt_level, --define, and --include_path options are compiler options. The --library, --heap_size, and --output_file options are linker options.

```
cl6x -mv6740 --c99 --opt_level=1 --define=c6748 --include_path="C:/ti/ti-cgt-c6000_8.3/include"
    hello.c objects.cpp algs.asm
    --run_linker --library=lnk.cmd --heap_size=0x800 --output_file=myprogram.out
```

Chapter 3

Using the C/C++ Compiler

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

3.1 About the Compiler

The compiler lets you compile, optimize, 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, assembly code, and linear assembly code. It 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 3.3.10 for more information.
- The **linker** combines object files to create a static executable file. The link step is optional, so you can compile and assemble many modules independently and link them later. See Chapter 6 for information about linking the files.

Note

Invoking the Linker

By default, the compiler does not invoke the linker. You can invoke the linker by using the --run_linker (-z) compiler option. See Section 6.1.1 for details.

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

3.2 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 3-6 through Table 3-28.
filenames	One or more C/C++ source files, assembly language source files , and linear assembly files.
--run_linker (**-z**)	Option that invokes the linker. The --run_linker option's short form is -z. See Chapter 6 for more information.
link_options	Options that control the linking process.
object files	Names of the 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 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, and link to create an executable program called myprogram.out, you will enter:

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

3.3 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 a help screen summary of the options, enter **cl6x** with no parameters on the command line.

The following apply to the compiler options:

- There are typically two ways of specifying a given option. The "long form" uses a two hyphen prefix and is usually a more descriptive name. The "short form" uses a single hyphen prefix and a combination of letters and numbers that are not always intuitive.
- Options are usually case sensitive.
- Individual options cannot be combined.
- An option with a 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*. Likewise, the option to specify the maximum amount of optimization can be expressed as -O=3. You can also specify a parameter directly after certain options, for example -O3 is the same as -O=3. 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 compiler options and precede any linker 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 3.4.1.

Table 3-1 through Table 3-28 summarize all options (including link options). Use the references in the tables for more complete descriptions of the options.

Table 3-1. Processor Options

Option	Alias	Effect	Section
--silicon_version=*id*	-mv	Selects target version. Defaults to 6400+. The other supported options are 6600 and 6740.	Section 3.3.5
--big_endian	-me	Produces object code in big-endian format.	Section 3.3.4

Table 3-2. Optimization Options(1)

Option	Alias	Effect	Section
--opt_level=off		Disables all optimization (default).	Section 4.1
--opt_level=*n*	-O*n*	Level 0 (-O0) optimizes register usage only. Level 1 (-O1) uses Level 0 optimizations and optimizes locally. Level 2 (-O2) uses Level 1 optimizations and optimizes globally. Level 3 (-O3) uses Level 2 optimizations and optimizes the file.	Section 4.1, Section 4.3
--opt_for_space=*n*	-ms	Controls code size on four levels (0, 1, 2, and 3).	Section 4.9
--opt_for_speed[=*n*]	-mf	Controls the tradeoff between size and speed (0-5 range). If this option is not specified or is specified without *n*, the default value is 4.	Section 4.2

(1) **Note:** Machine-specific options (see Table 3-12) can also affect optimization.

Table 3-3. Advanced Optimization Options(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 4.5

Table 3-3. Advanced Optimization Options[1] (continued)

Option	Alias	Effect	Section
--call_assumptions=*n*	-op*n*	Level 0 (-op0) specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler. Level 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. Level 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). Level 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 4.4.1
--disable_inlining		Prevents any inlining from occurring.	Section 3.11
--fp_mode={relaxed|strict}		Enables or disables relaxed floating-point mode.	Section 3.3.3
--fp_reassoc={on|off}		Enables or disables the reassociation of floating-point arithmetic.	Section 3.3.3
--fp_single_precision_constant		Causes all unsuffixed floating-point constants to be treated as single precision values instead of as double-precision constants.	Section 3.3.3
--gen_opt_info=*n*	-on*n*	Level 0 (-on0) disables the optimization information file. Level 1 (-on2) produces an optimization information file. Level 2 (-on2) produces a verbose optimization information file.	Section 4.3.1
--optimizer_interlist	-os	Interlists optimizer comments with assembly statements.	Section 4.16
--program_level_compile	-pm	Combines source files to perform program-level optimization.	Section 4.4
--sat_reassoc={on|off}		Enables or disables the reassociation of saturating arithmetic. Default is --sat_reassoc=off.	Section 3.3.3
--aliased_variables	-ma	Notifies the compiler that addresses passed to functions may be modified by an alias in the called function.	Section 4.12.1

(1) **Note:** Machine-specific options (see Table 3-12) can also affect optimization.

Table 3-4. Debug Options

Option	Alias	Effect	Section
--symdebug:dwarf	-g	Default behavior. Enables symbolic debugging. The generation of debug information does not impact optimization. Therefore, generating debug information is enabled by default.	Section 3.3.6 Section 4.17
--symdebug:dwarf_version=2|3		Specifies the DWARF format version.	Section 3.3.6
--symdebug:none		Disables all symbolic debugging.	Section 3.3.6 Section 4.17
--disable_push_pop		Disables the code-size optimization that calls the RTS functions _push_rts() and _pop_rts(). You may want to use this option if you receive warnings about calls to RTS routines that are placed out of range of the calling location.	--
--machine_regs		Displays reg operands as machine registers in assembly code.	Section 3.3.12

Table 3-5. Include Options

Option	Alias	Effect	Section
--include_path=*directory*	`-I`	Adds the specified directory to the #include search path.	Section 3.5.2.1
--preinclude=*filename*		Includes *filename* at the beginning of compilation.	Section 3.3.3

Table 3-6. Control Options

Option	Alias	Effect	Section
--compile_only	-c	Disables linking (negates --run_linker).	Section 6.1.3
--help	-h	Prints (on the standard output device) a description of the options understood by the compiler.	Section 3.3.2
--run_linker	-z	Causes the linker to be invoked from the compiler command line.	Section 3.3.2

Table 3-6. Control Options (continued)

Option	Alias	Effect	Section
--skip_assembler	-n	Compiles C/C++ source file or linear assembly source file, producing an assembly language output file. The assembler is not run and no object file is produced.	Section 3.3.2

Table 3-7. Language Options

Option	Alias	Effect	Section
--c89		Processes C files according to the ISO C89 standard.	Section 7.13
--c99		Processes C files according to the ISO C99 standard.	Section 7.13
--c11		Processes C files according to the ISO C11 standard.	Section 7.13
--c++14		Processes C++ files according to the ISO C++14 standard. The --c++03 option has been deprecated.	Section 7.13
--cpp_default	-fg	Processes all source files with a C extension as C++ source files.	Section 3.3.8
--exceptions		Enables C++ exception handling.	Section 7.6
--extern_c_can_throw		Allow extern C functions to propagate exceptions.	--
--float_operations_allowed ={none|all|32|64}		Restricts the types of floating point operations allowed.	Section 3.3.3
--gen_cross_reference_listing	-px	Generates a cross-reference listing file (.crl).	Section 3.9
--multithread		Inserts a build attribute into the compiler-generated object file that will cause the TI linker to choose a thread-safe version of the RTS library when auto-selecting an RTS library or resolving a reference to libc.a. Alternately, a linker option with the same name (--multithread) can be used to force the linker to choose a thread-safe version of the RTS library, even if none of the object files contain this build attribute. If you use the --openmp option, the --multithread option is enabled automatically.	Section 8.10.2
--openmp	--omp	Enables support for OpenMP. Using this option automatically enables the --multithread option, which causes the TI linker to choose a thread-safe version of the RTS library when auto-selecting an RTS library or resolving a reference to libc.a.	Section 8.10.1
--pending_instantiations=#		Specify the number of template instantiations that may be in progress at any given time. Use 0 to specify an unlimited number.	Section 3.3.4
--printf_support={nofloat|full| minimal}		Enables support for smaller, limited versions of the printf function family (sprintf, fprintf, etc.) and the scanf function family (sscanf, fscanf, etc.) run-time-support functions.	Section 3.3.3
--relaxed_ansi	-pr	Enables relaxed mode; ignores strict ISO violations. This is on by default. To disable this mode, use the --strict_ansi option.	Section 7.13.3
--rtti	-rtti	Enables C++ run-time type information (RTTI).	–-
--strict_ansi	-ps	Enables strict ANSI/ISO mode (for C/C++, not for K&R C). In this mode, language extensions that conflict with ANSI/ISO C/C++ are disabled. In strict ANSI/ISO mode, most ANSI/ISO violations are reported as errors. Violations that are considered discretionary may be reported as warnings instead.	Section 7.13.3
--vectypes={on|off}		Enable support for TI vector data types.	Section 7.3.2
--wchar_t={32|16}		Sets the size of the C/C++ type wchar_t. Default is 16 bits.	Section 3.3.4

Table 3-8. Parser Preprocessing Options

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 3.5.8
--preproc_includes[=*filename*]	-ppi	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive.	Section 3.5.9
--preproc_macros[=*filename*]	-ppm	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 3.5.10

Table 3-8. Parser Preprocessing Options (continued)

Option	Alias	Effect	Section
--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 3.5.4
--preproc_with_comment	-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 3.5.6
--preproc_with_compile	-ppa	Continues compilation after preprocessing with any of the -pp<x> options that normally disable compilation.	Section 3.5.5
--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 3.5.7

Table 3-9. Predefined Macro Options

Option	Alias	Effect	Section
--define=*name*[=*def*]	-D	Predefines *name*.	Section 3.3.2
--undefine=*name*	-U	Undefines *name*.	Section 3.3.2

Table 3-10. Diagnostic Message Options

Option	Alias	Effect	Section
--compiler_revision		Prints out the compiler release revision and exits.	--
--diag_error=*num*	-pdse	Categorizes the diagnostic identified by *num* as an error.	Section 3.7.1
--diag_remark=*num*	-pdsr	Categorizes the diagnostic identified by *num* as a remark.	Section 3.7.1
--diag_suppress=*num*	-pds	Suppresses the diagnostic identified by *num*.	Section 3.7.1
--diag_warning=*num*	-pdsw	Categorizes the diagnostic identified by *num* as a warning.	Section 3.7.1
--diag_wrap={on|off}		Wrap diagnostic messages (default is on). Note that this command-line option cannot be used within the Code Composer Studio IDE.	
--display_error_number	-pden	Displays a diagnostic's identifiers along with its text. Note that this command-line option cannot be used within the Code Composer Studio IDE.	Section 3.7.1
--emit_warnings_as_errors	-pdew	Treat warnings as errors.	Section 3.7.1
--issue_remarks	-pdr	Issues remarks (non-serious warnings).	Section 3.7.1
--no_warnings	-pdw	Suppresses diagnostic warnings (errors are still issued).	Section 3.7.1
--quiet	-q	Suppresses progress messages (quiet).	--
--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 3.7.1
--super_quiet	-qq	Super quiet mode.	--
--tool_version	-version	Displays version number for each tool.	--
--verbose		Display banner and function progress information.	--
--verbose_diagnostics	-pdv	Provides verbose diagnostic messages that display the original source with line-wrap. Note that this command-line option cannot be used within the Code Composer Studio IDE.	Section 3.7.1
--write_diagnostics_file	-pdf	Generates a diagnostic message information file. Compiler only option. Note that this command-line option cannot be used within the Code Composer Studio IDE.	Section 3.7.1

Table 3-11. Supplemental Information Options

Option	Alias	Effect	Section
--gen_preprocessor_listing	-pl	Generates a raw listing file (.rl).	Section 3.10
--section_sizes={on|off}		Generates section size information, including sizes for sections containing executable code and constants, constant or initialized data (global and static variables), and uninitialized data. (Default is off if this option is not included on the command line. Default is on if this option is used with no value specified.)	Section 3.7.1

Table 3-12. Run-Time Model Options

Option	Alias	Effect	Section
--assume_control_regs_read		Assume the FP and SAT bits are read.	Section 3.3.4
--common={on|off}		On by default. When on, uninitialized file scope variables are emitted as common symbols. When off, common symbols are not created.	Section 3.3.4
--debug_software_pipeline	-mw	Produce verbose software pipelining report.	Section 4.6.2
--disable_software_pipeline	-mu	Turns off software pipelining.	Section 4.6.1
--fp_not_associative	-mc	Prevents reordering of associative floating-point operations.	Section 4.13
--gen_data_subsections={on|off}		Place all aggregate data (arrays, structs, and unions) into subsections. This gives the linker more control over removing unused data during the final link step. See the link to the right for details about the default setting.	Section 6.2.3
--gen_func_subsections={on|off}	-mo	Puts each function in a separate subsection in the object file. If this option is not used, the default is off. See the link to the right for details about the default setting.	Section 6.2.2
--interrupt_threshold[=*num*]	-mi	Specifies an interrupt threshold value.	Section 3.12
--mem_model:const={far_aggregates|far|data}		Allows const objects to be made far independently of the --mem_model:data option.	Section 8.1.4.3
--mem_model:data={far_aggregates|near|far}		Determines data access model.	Section 8.1.4.1
--no_bad_aliases	-mt	Allows certain assumptions about aliasing and loops.	Section 4.12.2
--no_compress		Prevents compression.	--
--no_reload_errors		Turns off all reload-related loop buffer error messages.	--
--profile:breakpt		Enables breakpoint-based profiling.	Section 3.3.6 Section 4.17.1
--speculate_loads=*n*	-mh	Specifies speculative load byte count threshold. Allows speculative execution of loads with bounded address ranges.	Section 4.6.3.1
--speculate_unknown_loads		Allows speculative execution of loads with unbounded addresses.	Section 3.3.4
--use_const_for_alias_analysis	-ox	Uses const to disambiguate pointers.	Section 3.3.4

Table 3-13. Entry/Exit Hook Options

Option	Alias	Effect	Section
--entry_hook[=*name*]		Enables entry hooks.	Section 3.16
--entry_parm={none|name|address}		Specifies the parameters to the function to the --entry_hook option.	Section 3.16
--exit_hook[=*name*]		Enables exit hooks.	Section 3.16
--exit_parm={none|name|address}		Specifies the parameters to the function to the --exit_hook option.	Section 3.16
--remove_hooks_when_inlining		Removes entry/exit hooks for auto-inlined functions.	Section 3.16

Table 3-14. Feedback Options

Option	Alias	Effect	Section
--analyze={codecov|callgraph}		Generate analysis info from profile data.	Section 4.11.4.2
--analyze_only		Only generate analysis.	Section 4.11.4.2
--gen_profile_info		Generates instrumentation code to collect profile information.	Section 4.10.1.3
--use_profile_info=*file1*[, *file2*,...]		Specifies the profile information file(s).	Section 4.10.1.3

Table 3-15. Assembler Options

Option	Alias	Effect	Section
--keep_asm	-k	Keeps the assembly language (.asm) file.	Section 3.3.12
--asm_listing	-al	Generates an assembly listing file.	Section 3.3.12
--c_src_interlist	-ss	Interlists C source and assembly statements.	Section 3.13 Section 4.16

Table 3-15. Assembler Options (continued)

Option	Alias	Effect	Section
--src_interlist	-s	Interlists optimizer comments (if available) and assembly source statements; otherwise interlists C and assembly source statements.	Section 3.3.2
--asm_cross_reference_listing	-ax	Generates the cross-reference file.	Section 3.3.12
--asm_define=*name*[=*def*]	-ad	Sets the *name* symbol.	Section 3.3.12
--asm_dependency	-apd	Performs preprocessing; lists only assembly dependencies.	Section 3.3.12
--asm_includes	-api	Performs preprocessing; lists only included #include files.	Section 3.3.12
--asm_undefine=*name*	-au	Undefines the predefined constant *name*.	Section 3.3.12
--include_file=*filename*	-ahi	Includes the specified file for the assembly module.	Section 3.3.12
--no_const_clink		Stops generation of .clink directives for const global arrays.	Section 3.3.3
--strip_coff_underscore		Aids in transitioning hand-coded assembly from COFF to EABI.	Section 3.3.12

Table 3-16. File Type Specifier Options

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 3.3.8
--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 3.3.8
--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 3.3.8
--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 3.3.8
--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, including both *.c.obj and *.cpp.obj files.	Section 3.3.8

Table 3-17. Directory Specifier Options

Option	Alias	Effect	Section
--asm_directory=*directory*	-fs	Specifies an assembly file directory. By default, the compiler uses the current directory.	Section 3.3.11
--list_directory=*directory*	-ff	Specifies an assembly listing file and cross-reference listing file directory By default, the compiler uses the object file directory.	Section 3.3.11
--obj_directory=*directory*	-fr	Specifies an object file directory. By default, the compiler uses the current directory.	Section 3.3.11
--output_file=*filename*	-fe	Specifies a compilation output file name; can override --obj_directory.	Section 3.3.11
--pp_directory=*dir*		Specifies a preprocessor file directory. By default, the compiler uses the current directory.	Section 3.3.11
--temp_directory=*directory*	-ft	Specifies a temporary file directory. By default, the compiler uses the current directory.	Section 3.3.11

Table 3-18. Default File Extensions Options

Option	Alias	Effect	Section
--ap_extension=[.]*extension*	-el	Sets a default extension for linear assembly source files.	Section 3.3.10
--asm_extension=[.]*extension*	-ea	Sets a default extension for assembly source files.	Section 3.3.10
--c_extension=[.]*extension*	-ec	Sets a default extension for C source files.	Section 3.3.10
--cpp_extension=[.]*extension*	-ep	Sets a default extension for C++ source files.	Section 3.3.10
--listing_extension=[.]*extension*	-es	Sets a default extension for listing files.	Section 3.3.10
--obj_extension=[.]*extension*	-eo	Sets a default extension for object files.	Section 3.3.10

Table 3-19. Command Files Options

Option	Alias	Effect	Section
--cmd_file=*filename*	-@	Interprets contents of a file as an extension to the command line. Multiple -@ instances can be used.	Section 3.3.2

Table 3-20. Performance Advisor Options

Option	Alias	Effect	Section
--advice:performance[={all|none}]		Generates compiler optimization advice. Default is all.	Section 3.14
--advice:performance_file={stdout|stderr|user_specified_filename}		Specifies that advice be written to stdout, stderr, or a file.	Section 3.14
--advice:performance_dir={user_specified_directory_name}		Specifies that advice file be created in the named directory.	Section 3.14

3.3.1 Linker Options

The following tables list the linker options. See Chapter 6 of this document and the *TMS320C6000 Assembly Language Tools User's Guide* for details on these options.

Table 3-21. Linker Basic Options

Option	Alias	Description
--run_linker	-z	Enables linking.
--output_file=*file*	-o	Names the executable output file. The default filename is a .out file.
--map_file=*file*	-m	Produces a map or listing of the input and output sections, including holes, and places the listing in *file*.
--stack_size=*size*	[-]-stack	Sets C system stack size to *size* bytes and defines a global symbol that specifies the stack size. Default = 1K bytes.
--heap_size=*size*	[-]-heap	Sets heap size (for the dynamic memory allocation in C) to *size* bytes and defines a global symbol that specifies the heap size. Default = 1K bytes.

Table 3-22. File Search Path Options

Option	Alias	Description
--library=*file*	-l	Names an archive library or link command *file* as linker input.
--disable_auto_rts		Disables the automatic selection of a run-time-support library. See Section 6.3.1.1.
--priority	-priority	Satisfies unresolved references by the first library that contains a definition for that symbol.
--reread_libs	-x	Forces rereading of libraries, which resolves back references.
--search_path=*pathname*	`-I`	Alters library-search algorithms to look in a directory named with *pathname* before looking in the default location. This option must appear before the --library option.

Table 3-23. Command File Preprocessing Options

Option	Alias	Description
--define=*name*=*value*		Predefines *name* as a preprocessor macro.
--undefine=*name*		Removes the preprocessor macro *name*.
--disable_pp		Disables preprocessing for command files.

Table 3-24. Diagnostic Message Options

Option	Alias	Description
--diag_error=*num*		Categorizes the diagnostic identified by *num* as an error.
--diag_remark=*num*		Categorizes the diagnostic identified by *num* as a remark.
--diag_suppress=*num*		Suppresses the diagnostic identified by *num*.
--diag_warning=*num*		Categorizes the diagnostic identified by *num* as a warning.
--display_error_number		Displays a diagnostic's identifiers along with its text.
--emit_references:file[=*file*]		Emits a file containing section information. The information includes section size, symbols defined, and references to symbols.

Table 3-24. Diagnostic Message Options (continued)

Option	Alias	Description
--emit_warnings_as_errors	-pdew	Treat warnings as errors.
--issue_remarks		Issues remarks (non-serious warnings).
--no_demangle		Disables demangling of symbol names in diagnostic messages.
--no_warnings		Suppresses diagnostic warnings (errors are still issued).
--set_error_limit=*count*		Sets the error limit to *count*. The linker abandons linking after this number of errors. (The default is 100.)
--verbose_diagnostics		Provides verbose diagnostic messages that display the original source with line-wrap.
--warn_sections	-w	Displays a message when an undefined output section is created.

Table 3-25. Linker Output Options

Option	Alias	Description
--absolute_exe	-a	Produces an absolute, executable object file. This is the default; if neither --absolute_exe nor --relocatable is specified, the linker acts as if --absolute_exe were specified.
--ecc={ on | off }		Enable linker-generated Error Correcting Codes (ECC). The default is off.
--ecc:data_error		Inject specified errors into the output file for testing.
--ecc:ecc_error		Inject specified errors into the Error Correcting Code (ECC) for testing.
--mapfile_contents=*attribute*		Controls the information that appears in the map file.
--relocatable	-r	Produces a nonexecutable, relocatable output object file.
--xml_link_info=*file*		Generates a well-formed XML *file* containing detailed information about the result of a link.

Table 3-26. Symbol Management Options

Option	Alias	Description
--entry_point=*symbol*	-e	Defines a global symbol that specifies the primary entry point for the executable object file.
--globalize=*pattern*		Changes the symbol linkage to global for symbols that match *pattern*.
--hide=*pattern*		Hides symbols that match the specified *pattern*.
--localize=*pattern*		Make the symbols that match the specified *pattern* local.
--make_global=*symbol*	-g	Makes *symbol* global (overrides -h).
--make_static	-h	Makes all global symbols static.
--no_symtable	-s	Strips symbol table information and line number entries from the executable object file.
--retain={*symbol*| *section specification*}		Specifies a symbol or section to be retained by the linker.
--scan_libraries	-scanlibs	Scans all libraries for duplicate symbol definitions.
--symbol_map=*refname*=*defname*		Specifies a symbol mapping; references to the *refname* symbol are replaced with references to the *defname* symbol.
--undef_sym=*symbol*	-u	Adds *symbol* to the symbol table as an unresolved symbol.
--unhide=*pattern*		Excludes symbols that match the specified *pattern* from being hidden.

Table 3-27. Run-Time Environment Options

Option	Alias	Description
--arg_size=*size*	--args	Reserve *size* bytes for the argc/argv memory area.
--cinit_compression[=type]		Specifies the type of compression to apply to the C auto initialization data. Default is rle.
--copy_compression[=type]		Compresses data copied by linker copy tables. Default is rle.
--fill_value=*value*	-f	Sets default fill value for holes within output sections
--multithread		Causes the linker to choose a thread-safe version of the RTS library when auto-selecting an RTS library or resolving a reference to libc.a, even if none of the input object files contain the TI build attribute placed by the --multithread compiler option. If you used the --openmp compiler option to create any of the object files, the --multithread option is enabled automatically.

Table 3-27. Run-Time Environment Options (continued)

Option	Alias	Description
--ram_model	-cr	Initializes variables at load time. See Section 6.3.4 for details.
--rom_model	-c	Autoinitializes variables at run time. See Section 6.3.4 for details.
--trampolines[=off|on]		Generates far call trampolines. Default is on.

Table 3-28. Miscellaneous Options

Option	Alias	Description
--compress_dwarf[=off|on]		Aggressively reduces the size of DWARF information from input object files. Default is on.
--linker_help	[-]-help	Displays information about syntax and available options.
--minimize_trampolines[=off|postorder]		Places sections to minimize number of far trampolines required. Default is postorder.
--preferred_order=*function*		Prioritizes placement of functions.
--trampoline_min_spacing=*size*		When trampoline reservations are spaced more closely than the specified limit, tries to make them adjacent.
--unused_section_elimination[=off|on]		Eliminates sections that are not needed in the executable module. Default is on.
--zero_init=[off|on]		Controls preinitialization of uninitialized variables. Default is on. Always off if --ram_model is used.

3.3.2 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 4.16. 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. Use this option to avoid limitations on command line length or C style comments imposed by the operating system. Use a # or ; at the beginning of a line in the command file to include comments. You can add comments by surrounded by /* 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 example, the following indicates 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 6.1.3.

--define=*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, *name* is set to 1. This 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*="\"*string def*\"". For example, --define=car="\"sedan\""
- For UNIX, use --define=*name*="'*string def*"'. For example, --define=car="'sedan"'
- For CCS, enter the definition in a file and include that file with the --cmd_file option.

--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 3.5.2.1.

--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. This option's short form is -k.

--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 6.1.

--skip_assembler Compiles only. The specified source files are compiled but not assembled or linked. This 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* Undefines the predefined constant *name*. This option overrides any --define options for the specified constant. The --undefine option's short form is -U.

--verbose Displays progress information and toolset version while compiling. Resets the --quiet option.

3.3.3 Miscellaneous Useful Options

Following are detailed descriptions of miscellaneous options:

--float_operations_allowed={none|all|32|64} Restricts types of floating point operations allowed. The default is all. If set to none, 32, or 64, the application is checked for operations performed at runtime. For example, if --float_operations_allowed=32 is specified on the command line, the compiler issues an error if a double precision operation will be generated. This can be used to ensure that double precision operations are not accidentally introduced into an application. The checks are performed after relaxed mode optimizations have been performed, so illegal operations that are completely removed result in no diagnostic messages.

--fp_mode={relaxed|strict} The default floating-point mode is strict. To enable relaxed floating-point mode use the --fp_mode=relaxed option. Relaxed floating-point mode causes double-precision floating-point computations and storage to be converted to single-precision floating-point where possible. This behavior does not conform with ISO, but it results in faster code, with some loss in accuracy. The following specific changes occur in relaxed mode:

- If a double-precision floating-point expression's result is assigned to a single-precision floating-point, an integer, or immediately used in a single-precision context, the expression's computations are converted to single-precision computations. Double-precision constants in the expression are converted to single-precision if they can be correctly represented as single-precision constants.
- Calls to double-precision functions in math.h are converted to their single-precision counterparts if all arguments are single-precision and the result is used in a single-precision context. The math.h header file must be included for this optimization to work.
- Division by a constant is converted to inverse multiplication.

In the following examples, iN=integer variable, fN=float variable, and 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 1.1f
```

To enable relaxed floating-point mode use --fp_mode=relaxed, which also sets --fp_reassoc=on. To disable relaxed floating-point mode use --fp_mode=strict, which also sets --fp_reassoc=off.

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 --strict_ansi is set, --fp_reassoc=off is set since reassociation of floating-point arithmetic is an ANSI violation.

Because floating-point values are of limited precision, and because floating-point operations round, floating-point arithmetic is neither associative nor distributive. For instance, (1 + 3e100) - 3e100 is not equal to 1 + (3e100 - 3e100). If strictly following IEEE 754, the compiler cannot, in general, reassociate floating-point operations. Using --fp_reassoc=on allows the compiler to perform the algebraic reassociation, at the cost of a small amount of precision for some operations.

--fp_single_precision_constant Causes all unsuffixed floating-point constants to be treated as single precision values. By default, if this option is not used, such constants are implicitly converted to double-precision constants as expected for EABI output. If your floating-point constants always fit within the range supported for 32-bit floats, treating them as such can improve performance.
This option may be used with any settings for the --fp_mode and --float_support options.

--no_const_clink	Tells the compiler to not generate conditional linking (.clink) directives for const global arrays. By default, these arrays are placed in a .const subsection and conditionally linked.
--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.
--printf_support={full|nofloat|minimal}	Enables support for smaller, limited versions of the printf function family (sprintf, fprintf, etc.) and the scanf function family (sscanf, fscanf, etc.) run-time-support functions. The valid values are: • full: Supports all format specifiers. This is the default. • nofloat: Excludes support for printing and scanning floating-point values. Supports all format specifiers except %a, %A, %f, %F, %g, %G, %e, and %E. • minimal: Supports the printing and scanning of integer, char, or string values without width or precision flags. Specifically, only the %%, %d, %o, %c, %s, and %x format specifiers are supported. There is no run-time error checking to detect if a format specifier is used for which support is not included. The --printf_support option precedes the --run_linker option, and must be used when performing the final link.
--sat_reassoc={on|off}	Enables or disables the reassociation of saturating arithmetic.

3.3.4 Run-Time Model Options

These options are specific to the TMS302C6000 toolset. See the referenced sections for more information. TMS320C6000-specific assembler options are listed in Section 3.3.12.

The C6000 compiler now supports only the Embedded Application Binary Interface (EABI) ABI, which uses the ELF object format and the DWARF debug format. Refer to the *C6000 Embedded Application Binary Interface Application Report* (SPRAB89) for details about EABI. If you want support for the legacy COFF ABI, please use the C6000 v7.4.x Code Generation Tools and refer to SPRU187 and SPRU186 for documentation.

--assume_control_regs_read	Tells the compiler to assume that the FP and SAT control bits are read somewhere in the program. As a result, the compiler does not speculate instructions that may set the FP or SAT control bits. See Section 8.6.15 for more information.
--advice:performance	Generates compile-time optimization advice. See Section 3.14.
--big_endian	Produces code in big-endian format. By default, little-endian code is produced.
--common={on|off}	When on (the default), uninitialized file scope variables are emitted as common symbols. When off, common symbols are not created. The benefit of allowing common symbols to be created is that generated code can remove unused variables that would otherwise increase the size of the .bss section. (Uninitialized variables of a size larger than 32 bytes are separately optimized through placement in separate subsections that can be omitted from a link.) Variables cannot be common symbols if they are assigned to a section other than .bss or have a specified memory bank.
--debug_software_pipeline	Produces verbose software pipelining report. See Section 4.6.2.
--disable_software_pipeline	Turns off software pipelining. See Section 4.6.1.
--fp_not_associative	Compiler does not reorder floating-point operations. See Section 4.13.
--interrupt_threshold=*n*	Specifies an interrupt threshold value *n* that sets the maximum cycles the compiler can disable interrupts. See Section 3.12.
--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 8.1.4.3
--mem_model:data=*type*	Specifies data access model as *type* far, far_aggregates, or near. Default is far_aggregates. See Section 8.1.4.1.
--pending_instantiations=#	Specify the number of template instantiations that may be in progress at any given time. Use 0 to specify an unlimited number.
--silicon_version=*num*	Selects the target CPU version. See Section 3.3.5.
--speculate_loads=*n*	Specifies speculative load byte count threshold. Allows speculative execution of loads with bounded addresses. See Section 4.6.3.1.
--speculate_unknown_loads	Allows speculative execution of loads with unbounded addresses.
--static_template_instantiation	Instantiates all template entities in the current file as needed though the parser. These instantiations are also given internal (static) linkage. This option may provide a slight improvement to compilation speed.

--use_const_for_alias_analysis	Uses const to disambiguate pointers.
--wchar_t={32|16}	Sets the size (in bits) of the C/C++ type wchar_t. By default the compiler generates 16-bit wchar_t. 16-bit wchar_t objects are not compatible with 32-bit wchar_t objects; an error is generated if they are combined. When the --linux option is specified, it implies --wchar_t=32 since Linux uses 32-bit extended characters.

3.3.5 Selecting Target CPU Version (--silicon_version Option)

The --silicon_version option controls the use of target-specific instructions and alignment. The alias for this option is -mv. If this option is not used, the compiler generates code for the C6400+ parts by default.

Specify the family of the part, for example, --silicon_version=6400+ or --silicon_version=6740.

Target CPU version options include:

- -mv6400+ or -mv64+
- -mv6740
- -mv6600

If you want support for C6200, C6400, C6700, or C6700+ targets, please use the C6000 v7.4.x Code Generation Tools and refer to SPRU187 and SPRU186 for documentation. These targets are no longer supported by the C6000 v8.x Code Generation Tools.

3.3.6 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.
--symdebug:dwarf	(Default) 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. See Section 4.17. For details on the DWARF 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 (the default) is specified. By default, the compiler generates DWARF version 3 debug information. 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.

3.3.7 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 .c.obj .cpp.obj .o* .dll .so	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 3.3.8. 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 3.3.11.

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.

3.3.8 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 source 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.

Note

The default file extensions for object files created by the compiler have been changed in order to prevent conflicts when C and C++ files have the same names. Object files generated from C source files have the .c.obj extension. Object files generated from C++ source files have the .cpp.obj extension.

3.3.9 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 3.3.10 for more information about filename extension conventions.

3.3.10 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
```

3.3.11 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:

--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
```

--output_file=*filename*
Specifies a compilation output file name; can override --obj_directory. For example:
```
cl6x --output_file=transfer
```

--pp_directory=*directory*
Specifies a preprocessor file directory for object files (default is .). For example:
```
cl6x --pp_directory=d:\preproc
```

--temp_directory=*directory*
Specifies a directory for temporary intermediate files. For example:
```
cl6x --temp_directory=d:\temp
```

3.3.12 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.*

--asm_define=*name*[=*def*]
Predefines the constant *name* for the assembler; produces a .set directive for a constant or an .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 name.

--asm_cross_reference_listing
Produces a symbolic cross-reference in the listing file.

--include_file=*filename*	Includes the specified file for the assembly module; acts like an .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. Compression changes 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.
--strip_coff_underscore	Aids in transitioning hand-coded assembly from COFF to EABI. Although the COFF output is no longer supported, this option remains available as a COFF ABI to ELF EABI migration aid. For COFF ABI, the compiler prepended an underscore to the beginning of all C/C++ identifiers. For EABI, the link-time symbol is the same as the C/C++ identifier name. This option removes the underscore prefix from legacy symbol references as needed.

3.4 Controlling the Compiler Through Environment Variables

An environment variable is a system symbol you define and assign a string to. Setting environment variables is useful if 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 device-specific environment variables instead.

3.4.1 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 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 repeatedly 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
```

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

3.4.2 Naming One or More Alternate Directories (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=`

3.5 Controlling the Preprocessor

This section describes 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.

3.5.1 Predefined Macro Names

The compiler maintains and recognizes the predefined macro names listed in Table 3-29.

Table 3-29. Predefined C6000 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.
__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.
__PTRDIFF_T_TYPE__	Defined to the type of ptrdiff_t.

Table 3-29. Predefined C6000 Macro Names (continued)

Macro Name	Description
__SIZE_T_TYPE__	Defined to the type of size_t.
__STDC__ (1)	Defined to 1 to indicate that compiler conforms to ISO C Standard. See Section 7.1 for exceptions to ISO C conformance.
__STDC_VERSION__	C standard macro.
__STDC_HOSTED__	C standard macro. Always defined to 1.
__STDC_NO_THREADS__	C standard macro. Always defined to 1.
__TI_32BIT_LONG__	Defined to 1 if the type "long" is 32 bits wide; otherwise, it is undefined.
__TI_40BIT_LONG__	Defined to 1 if __TI_32BIT_LONG__ is not defined; otherwise, it is undefined.
__TI_C99_COMPLEX_ENABLED__	Defined to 1 if complex data types are enabled. This is always the case, though math operations are available only if complex.h is included.
__TI_COMPILER_VERSION__	Defined to a 7-9 digit integer, depending on if X has 1, 2, or 3 digits. 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.
__TI_EABI__	Always defined to 1.
__TI_GNU_ATTRIBUTE_SUPPORT__	Defined to 1 if GCC extensions are enabled (which is the default)
__TI_STRICT_ANSI_MODE__	Defined to 1 if strict ANSI/ISO mode is enabled (the --strict_ansi option is used); otherwise, it is defined as 0.
__TI_STRICT_FP_MODE__	Defined to 1 if --fp_mode=strict is used (default); otherwise, it is defined as 0.
__TI_WCHAR_T_BITS__	Defined to the type of wchar_t.
__TIME__ (1)	Expands to the compilation time in the form "*hh:mm:ss*"
_TMS320C6X	Always defined
_TMS320C6400_PLUS	Defined if target is C6400+, C6740, or C6600
_TMS320C6740	Defined if target is C6740 or C6600
_TMS320C6600	Defined if target is C6600
__TMS320C6X__	Always defined for use as alternate name for _TMS320C6x
__WCHAR_T_TYPE__	Defined to the type of wchar_t.

(1) Specified by the ISO standard

You can use the names listed in Table 3-29 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");
```

3.5.2 The Search Path for #include Files

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

- If you enclose the filename in double quotes (" "), the compiler searches for the file in 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 3.5.2.1 for information on using the --include_path option. See Section 3.4.2 for more information on input file directories.

3.5.2.1 Adding a Directory to 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 C6X_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
```

3.5.3 Support for the #warning and #warn Directives

In strict ANSI mode, the TI preprocessor allows you to use the #warn directive to cause the preprocessor to issue a warning and continue preprocessing. The #warn directive is equivalent to the #warning directive supported by GCC, IAR, and other compilers.

If you use the --relaxed_ansi option (on by default), both the #warn and #warning preprocessor directives are supported.

3.5.4 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.

The --preproc_only option is useful when creating a source file for a technical support case or to ask a question about your code. It allows you to reduce the test case to a single source file, because #include files are incorporated when the preprocessor runs.

3.5.5 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.

3.5.6 Generating a Preprocessed Listing File with Comments (--preproc_with_comment Option)

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

3.5.7 Generating Preprocessed Listing with Line-Control Details (--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.

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

The --preproc_dependency option performs preprocessing only. Instead of writing preprocessed output, it 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 a .pp extension.

3.5.9 Generating a List of Files Included with #include (--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.

3.5.10 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.

The output includes only those files directly included by the source file. Predefined macros are listed first and indicated by the comment /* Predefined */. User-defined macros are listed next and indicated by the source filename.

3.6 Passing Arguments to main()

Some programs pass arguments to main() via argc and argv. This presents special challenges in an embedded program that is not run from the command line. In general, argc and argv are made available to your program through the .args section. There are various ways to populate this section for use by your program.

To cause the linker to allocate an .args section of the appropriate size, use the --arg_size=*size* linker option. This option tells the linker to allocate an uninitialized section named .args, which can be used by the loader to pass arguments from the command line of the loader to the program. The *size* is the number of bytes to be allocated. When you use the --arg_size option, the linker defines the __c_args__ symbol to contain the address of the .args section.

It is the responsibility of the loader to populate the .args section. The loader and the target boot code can use the .args section and the __c_args__ symbol to determine whether and how to pass arguments from the host to the target program. The format of the arguments is an array of pointers to char on the target. Due to variations in loaders, it is not specified how the loader determines which arguments to pass to the target.

If you are using Code Composer Studio to run your application, you can use the Scripting Console tool to populate the .args section. To open this tool, choose **View > Scripting Console** from the CCS menus. You can use the loadProg command to load an object file and its associated symbol table into memory and pass an array of arguments to main(). These arguments are automatically written to the allocated .args section.

The loadProg syntax is as follows, where *file* is an executable file and *args* is an object array of arguments. Use JavaScript to declare the array of arguments before using this command.

```
loadProg(file, args)
```

The .args section is loaded with the following data for non-SYS/BIOS-based executables, where each element in the argv[] array contains a string corresponding to that argument:

```
Int argc;
Char * argv[0];
Char * argv[1];
...
Char * argv[n];
```

For SYS/BIOS-based executables, the elements in the .args section are as follows:

```
Int argc;
Char ** argv;    /* points to argv[0] */
Char * envp;     /* ignored by loadProg command */
Char * argv[0];
Char * argv[1];
...
Char * argv[n];
```

For more details, see the "Scripting Console" page.

3.7 Understanding Diagnostic Messages

One of the primary functions of the compiler and linker is to report diagnostic messages for the source program. A diagnostic message indicates that something may be wrong with the program. 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 a 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 may continue, but object code is not generated.
- A **warning** indicates something that is likely to be a problem, but cannot be proven to be an error. For example, the compiler emits a warning for an unused variable. An unused variable does not affect program execution, but its existence suggests that you might have meant to use it. Compilation continues and object code is generated (if no errors are detected).
- A **remark** is less serious than a warning. It may indicate something that is a potential problem in rare cases, or the remark may be strictly informational. 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.
- **Advice** provides information about recommended usage. See Section 4.15 for details.

Diagnostic messages 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 code line is not printed. Use the --verbose_diagnostics compiler option to display 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 diagnostic messages 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 errors are determined to be discretionary based on the severity in 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.

3.7.1 Controlling Diagnostic Messages

The C/C++ compiler provides diagnostic options to control compiler- and linker-generated diagnostic messages. 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 diagnostic messages.

--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 diagnostic messages.

--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 diagnostic messages.

--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 diagnostic messages.

--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 3.7.

--emit_warnings_as_errors Treats all warnings as errors. This option cannot be used with the --no_warnings option. The --diag_remark option takes precedence over this option. This option takes precedence over the --diag_warning option.

--issue_remarks Issues remarks (non-serious warnings), which are suppressed by default.

--no_warnings Suppresses diagnostic warnings (errors are still issued).

--section_sizes={on|off} Generates section size information, including sizes for sections containing executable code and constants, constant or initialized data (global and static variables), and uninitialized data. Section size information is output during both the assembly and linking phases. This option should be placed on the command line with the compiler options (that is, before the --run_linker or --z option).

--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 diagnostic messages that display the original source with line-wrap and indicate the position of the error in the source line. Note that this command-line option cannot be used within the Code Composer Studio IDE.

--write_diagnostics_file Produces a diagnostic message information file with the same source file name with an *.err* extension. (The --write_diagnostics_file option is not supported by the linker.) Note that this command-line option cannot be used within the Code Composer Studio IDE.

3.7.2 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 produces no diagnostic messages (because remarks are disabled by default).

Note

You can suppress any non-fatal errors, but be careful to make sure you only suppress diagnostic messages that you understand and are known not to affect the correctness of your program.

3.8 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.

3.9 Generating Cross-Reference Listing Information (--gen_cross_reference_listing Option)

The --gen_cross_reference_listing option generates a cross-reference listing file that contains reference information for each identifier in the source file. The listing file describes where each symbol is referenced and defined.

A cross-reference listing file with a .crl extension is generated for every source file. The files have the same name as their corresponding source file. (The --gen_cross_reference_listing option is separate from --asm_cross_reference_listing, which is an assembler rather than a compiler option.)

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	

3.10 Generating a Raw Listing File (--gen_preprocessor_listing Option)

The --gen_preprocessor_listing 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
- Diagnostic messages
- 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 3-30.

Table 3-30. 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_preprocessor_listing option also includes diagnostic identifiers as defined in Table 3-31.

Table 3-31. 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 3-31 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

Diagnostic messages 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 3.7.

3.11 Using Inline Function Expansion

When an inline function is called, a copy of the C/C++ source code for the function is inserted at the point of the call. This is known as inline function expansion, commonly called *function inlining* or just *inlining*. Inline function expansion can speed up execution by eliminating function call overhead. This is particularly beneficial for very small functions that are called frequently. Function inlining involves a tradeoff between execution speed and code size, because the code is duplicated at each function call site. Large functions that are called in many places are poor candidates for inlining.

Note

Excessive Inlining Can Degrade Performance: Excessive inlining can make the compiler dramatically slower and degrade the performance of generated code.

Function inlining is triggered by the following situations:

- The use of built-in intrinsic operations. Intrinsic operations look like function calls, and are inlined automatically, even though no function body exists.
- Use of the `inline` keyword or the equivalent `__inline` keyword. Functions declared with the inline keyword may be inlined by the compiler if you set --opt_level=0 or greater. The inline keyword is a suggestion from the programmer to the compiler. Even if your optimization level is high, inlining is still optional for the compiler. The compiler decides whether to inline a function based on the length of the function, the number of times it is called, your --opt_for_speed setting, and any contents of the function that disqualify it from inlining (see Section 3.11.2). Functions can be inlined at --opt_level=0 or above if the function body is visible in the same module or if -pm is also used and the function is visible in one of the modules being compiled. Functions defined as both static and inline are more likely to be inlined.
- When --opt_level=3 is used, the compiler may automatically inline eligible functions even if they are not declared as inline functions. The same list of decision factors listed for functions explicitly defined with the inline keyword is used. For more about automatic function inlining, see Section 4.5.
- The pragma FUNC_ALWAYS_INLINE (Section 7.9.10) and the equivalent `always_inline` attribute (Section 7.14.2) force a function to be inlined (where it is legal to do so) unless --opt_level=off. That is, the pragma FUNC_ALWAYS_INLINE forces function inlining even if the function is not declared as inline and the --opt_level=0 or --opt_level=1.
- The FORCEINLINE pragma (Section 7.9.8) forces functions to be inlined in the annotated statement. That is, it has no effect on those functions in general, only on function calls in a single statement. The FORCEINLINE_RECURSIVE pragma forces inlining not only of calls visible in the statement, but also in the inlined bodies of calls from that statement.
- The --disable_inlining option prevents any inlining. The pragma FUNC_CANNOT_INLINE prevents a function from being inlined. The NOINLINE pragma prevents calls within a single statement from being inlined. (NOINLINE is the inverse of the FORCEINLINE pragma.)

Note

Function Inlining Can Greatly Increase Code Size: Function inlining 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.

The semantics of the `inline` keyword in C code follow the C99 standard. The semantics of the `inline` keyword in C++ code follow the C++ standard.

The `inline` keyword is supported in all C++ modes, in relaxed ANSI mode for all C standards, and in strict ANSI mode for C99 and C11. It is disabled in strict ANSI mode for C89, because it is a language extension that could conflict with a strictly conforming program. If you want to define inline functions while in strict ANSI C89 mode, use the alternate keyword `__inline`.

Compiler options that affect inlining are: --opt_level, --auto_inline, --remove_hooks_when_inlining, --opt_for_speed, and --disable_inlining.

3.11.1 Inlining Intrinsic Operators

The compiler has a number of built-in function-like operations called intrinsics. The implementation of an intrinsic function is handled by the compiler, which substitutes a sequence of instructions for the function call. This is similar to the way inline functions are handled; however, because the compiler knows the code of the intrinsic function, it can perform better optimization.

Intrinsics are inlined whether or not you use the optimizer.

For details about intrinsics, and a list of the intrinsics, see Section 8.6.6. In addition to those listed, abs and memcpy are implemented as intrinsics.

3.11.2 Inlining Restrictions

The compiler makes decisions about which functions to inline based on the factors mentioned in Section 3.11. In addition, there are several restrictions that can disqualify a function from being inlined by automatic inlining or inline keyword-based inlining.

The compiler will leave calls as they are if the function:

- Has a different number of arguments than the call site
- Has an argument whose type is incompatible with the corresponding call site argument
- Is not declared inline and returns void but its return value is needed

The compiler will also not inline a call if the function has features that create difficult situations for the compiler:

- Has a variable-length argument list
- Never returns
- Is a recursive or non-leaf function that exceeds the depth limit
- Is not declared inline and contains an asm() statement that is not a comment
- Is an interrupt function
- Is the main() function
- Is not declared inline and will require too much stack space for local array or structure variables
- Contains a volatile local variable or argument
- Is a C++ function that contains a catch
- Is not defined in the current compilation unit

A call in a statement that is annotated with a NOINLINE pragma will not be inlined, regardless of other indications (including a FUNC_ALWAYS_INLINE pragma or always_inline attribute on the called function).

A call in a statement that is annotated with a FORCEINLINE pragma will always be inlined, if it is not disqualified for one of the reasons above, even if the called function has a FUNC_CANNOT_INLINE pragma or cannot_inline attribute.

In other words, a statement-level pragma overrides a function-level pragma or attribute. If both NOINLINE and FORCEINLINE apply to the same statement, the one that appears first is used and the rest are ignored.

3.11.3 Unguarded Definition-Controlled Inlining

The inline keyword causes a function to be expanded inline at the point where it is called rather than 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 to turn on definition-controlled inlining. Automatic inlining is also turned on when using --opt_level=3.

Example 3-1 uses the inline keyword. The function call is replaced by the code in the called function.

Example 3-1. 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);
   ...
}
```

3.11.4 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 in case 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 3-2.
- Create an identical version of the function definition in a .c or .cpp file, as shown in Example 3-3.

In the following examples there are two definitions of the strlen function. The first (Example 3-2), 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).

The second definition (see Example 3-3) 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 non-inline version of strlen's prototype.

Example 3-2. Header File string.h

```
/*****************************************************************************/
/* string.h  vx.xx   (Excerpted)                                             */
/*****************************************************************************/
#ifdef _INLINE
#define _IDECL static inline
#else
#define _IDECL extern _CODE_ACCESS
#endif
_IDECL size_t  strlen(const char *_string);
#ifdef _INLINE
/*****************************************************************************/
/*  strlen                                                                   */
/*****************************************************************************/
static inline size_t strlen(const char *string)
{
   size_t      n = (size_t)-1;
   const char *s = string - 1;
   do n++; while (*++s);
   return n
}
#endif
```

Example 3-3. Library Definition File

```
/*****************************************************************************/
/*  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;
}
```

3.12 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 7.9.13 or the *TMS320C6000 Programmer's Guide* for more information.

Note

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

The run-time-support library files provided with the compiler are not built with the interrupt flexibility option. Refer to the readme file to see how the run-time-support library files were built for your release. See Section 9.4 to build your own run-time-support library files with the interrupt flexibility option.

Note

Special Cases with the --interrupt_threshold Option

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.

3.13 Using Interlist

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

The easiest way to invoke the interlist feature is to use the --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.

For information about using the interlist feature with the optimizer, see Section 4.16. Using the --c_src_interlist option can cause performance and/or code size degradation.

The following example shows a typical interlisted assembly 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
```

3.14 Generating and Using Performance Advice

The compiler can do better optimization in some cases, if the user aids the compiler by providing additional information in the code. The compiler can prompt you to take certain actions to improve performance, by emitting "Advice". To get this Advice, use the --advice:performance option:

```
cl6x --advice:performance -o3 filename.c
```

This Performance Advice is of 3 different types :

- Advice to use correct compiler options
- Advice to prevent software pipeline disqualification
- Advice to improve loop performance

For more details on Using Performance Advice to Optimize your Code, see Section 4.15

3.15 About the Application Binary Interface

An Application Binary Interface (ABI) defines the low level interface between object files, and between an executable and its execution environment. An ABI allows ABI-compliant object files to be linked together, regardless of their origin, and allows the resulting executable to run on any system that supports that ABI.

The C6000 compiler supports only the Embedded Application Binary Interface (EABI) ABI, which works only with object files that use the ELF object file format and the DWARF debug format. If you want support for the legacy COFF ABI, please use the C6000 v7.4.x Code Generation Tools and refer to SPRU187 and SPRU186 for documentation.

There is no automated COFF to ELF conversion tool; you will need to recompile or reassemble assembly code. See the *TMS320C6000 EABI Migration Guide* (SPRAB90) for the benefits of migration from COFF to ELF and links to strategies and issues for migration.

3.16 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_parm{=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_parm{=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.

You can use the --remove_hooks_when_inlining option to remove entry/exit hooks for functions that are auto-inlined by the optimizer.

See Section 7.9.26 for information about the NO_HOOKS pragma.

This page intentionally left blank.

Chapter 4

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.

4.1 Invoking Optimization

The C/C++ compiler is able to perform various optimizations, which are performed by the optimizer and the code generator:

The *optimizer* performs high-level optimizations in the stand-alone optimization pass. Use higher optimization levels, such as --opt_level=2 and --opt_level=3, to achieve optimal code.

The *code generator* performs several additional optimizations. These are low-level, target-specific optimizations. It performs these regardless of whether you invoke the optimizer and are always enabled, though they are more effective when the optimizer is used.

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* as an alias for the --opt_level option. The *n* denotes the level of optimization (0, 1, 2, 3), which controls the type and degree of optimization.

- **--opt_level=off** or **-Ooff**
  - Performs no optimization
- **--opt_level=0** or **-O0**
  - Performs control-flow-graph simplification (Section 4.18.3)
  - Allocates variables to registers (Section 4.18.13)
  - Performs loop rotation (Section 4.18.10)
  - Eliminates unused code
  - Simplifies expressions and statements (Section 4.18.5)
  - Expands calls to functions declared as inline (Section 4.18.6)
- **--opt_level=1** or **-O1** Performs all --opt_level=0 (-O0) optimizations, plus:
  - Performs local copy/constant propagation (Section 4.18.4)
  - Removes unused assignments (Section 4.18.4)
  - Eliminates local common expressions
- **--opt_level=2** or **-O2** Performs all --opt_level=1 (-O1) optimizations, plus:
  - Performs software pipelining (Section 4.6)
  - Performs loop optimizations
  - Eliminates global common subexpressions (Section 4.18.4)
  - Eliminates global unused assignments (Section 4.18.4)
  - Converts array references in loops to incremented pointer form (Section 4.18.8)
  - Performs loop unrolling (Section 7.9.33)
- **--opt_level=3** or **-O3** Performs all --opt_level=2 (-O2) optimizations, plus:
  - Removes all functions that are never called (Section 4.4)
  - Simplifies functions with return values that are never used (Section 4.4)
  - Inlines calls to small functions (Section 3.11 and Section 4.5)
  - 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 (Section 4.4)
  - Performs other optimizations (Section 4.3 and Section 4.4)

For details about how the --opt_level and --opt_for_speed options and various pragmas affect inlining, see Section 3.11.

Debugging is enabled by default, and the optimization level is unaffected by the generation of debug information.

Note

Do Not Lower the Optimization Level to Control Code Size: 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 4.9.

Note

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

4.2 Controlling Code Size Versus Speed

To balance the tradeoff between code size and speed, use the --opt_for_speed option. The level of optimization (0-5) controls the type and degree of code size or code speed optimization:

- --opt_for_speed=0

  Optimizes code size with a *high* risk of worsening or impacting performance.
- --opt_for_speed=1

  Optimizes code size with a *medium* risk of worsening or impacting performance.
- --opt_for_speed=2

  Optimizes code size with a *low* risk of worsening or impacting performance.
- --opt_for_speed=3

  Optimizes code performance/speed with a *low* risk of worsening or impacting code size.
- --opt_for_speed=4

  Optimizes code performance/speed with a *medium* risk of worsening or impacting code size.
- --opt_for_speed=5

  Optimizes code performance/speed with a *high* risk of worsening or impacting code size.

If you specify the --opt_for_speed option without a parameter, the default setting is --opt_for_speed=4. If you do not specify the --opt_for_speed option, the default setting is 4

The older mechanism for controlling code space, the --opt_for_space option, has the following equivalences with the --opt_for_speed option:

--opt_for_space	--opt_for_speed
none	=4
=0	=3
=1	=2
=2	=1
=3	=0

4.3 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 4-1 work with --opt_level=3 to perform the indicated optimization:

Table 4-1. Options That You Can Use With --opt_level=3

If You ...	Use this Option	See
Want to create an optimization information file	--gen_opt_level=*n*	Section 4.3.1
Want to compile multiple source files	--program_level_compile	Section 4.4

Note

Do Not Lower the Optimization Level to Control Code Size

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.

4.3.1 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 4-2 to select the appropriate level to append to the option.

Table 4-2. 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_info=0
Want to produce an optimization information file	--gen_opt_info=1
Want to produce a verbose optimization information file	--gen_opt_info=2

4.4 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.

The --program_level_compile option requires use of --opt_level=3 in order to perform these optimizations.

To see which program-level optimizations the compiler is applying, use the --gen_opt_level=2 option to generate an information file. See Section 4.3.1 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.

4.4.1 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 4-3 to select the appropriate level to append to the --call_assumptions option.

Table 4-3. 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 4-4 lists the combinations of --call_assumptions levels and conditions that cause the compiler to revert to other --call_assumptions levels.

Table 4-4. Special Considerations When Using the --call_assumptions Option

If --call_assumptions 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* no interrupt functions are defined, *and* no functions are identified by the FUNC_EXT_CALLED pragma	Reverts to --call_assumptions=0
--call_assumptions=1 or --call_assumptions=2	A main function is defined, *or*, an interrupt function is defined, *or* a function is 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 4.4.2 for information about these situations.

4.4.2 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 7.9.12) 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 4.4.1 for information about the --call_assumptions=*n* option.

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.

  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.
- **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.

4.5 Automatic Inline Expansion (--auto_inline Option)

When optimizing with the --opt_level=3 option (aliased as -O3), 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.

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 automatically inlines small functions.

For information about interactions between command-line options, pragmas, and keywords that affect inlining, see Section 3.11.

Note

Some Functions Cannot Be Inlined: 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 3.11.2.

Note

Optimization Level 3 and Inlining: In order to turn on automatic inlining, you must use the --opt_level=3 option. If you desire the --opt_level=3 optimizations, but not automatic inlining, use --auto_inline=0 with the --opt_level=3 option.

Note

Inlining and Code Size: Expanding functions inline increases code size, especially inlining a function that is called in a number of places. Function inlining is optimal for functions that are called only from a small number of places and for small functions. To prevent increases in code size because of inlining, use the --auto_inline=0 option. This option causes the compiler to inline intrinsics only.

4.6 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 4.1.)

Figure 4-1 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*.

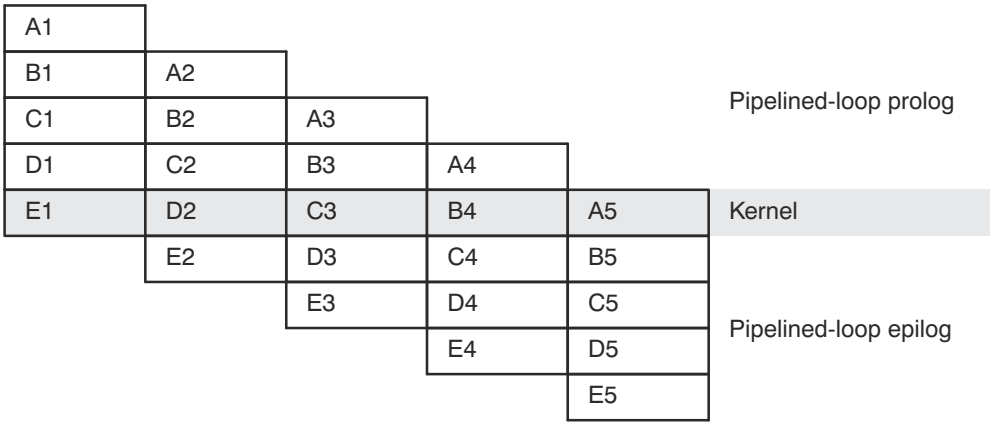


Figure 4-1. Software-Pipelined Loop

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*.

4.6.1 Turn Off Software Pipelining (--disable_software_pipeline 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_pipeline option affects both compiled C/C++ code and assembly optimized code.

Note

Software Pipelining May Increase Code Size

Software pipelining without the use of SPLOOP can lead to significant increases in code size. To control code size for loops that get software pipelined, the --opt_for_space option is recommended over the --disable_software_pipeline option. The --opt_for_space option is capable of disabling non-SPLOOP software pipelining if necessary to achieve code size savings, but it does not affect the SPLOOP capability (Section 4.8). SPLOOP does not significantly increase code size, but can greatly speed up a loop. Using the --disable_software_pipeline option disables all software pipelining including SPLOOP.

4.6.2 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 4-1 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.

Note

More Details on Software Pipelining Information

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 4-1. 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
;*
;*------------------------------------------------------------------------------*
```

4.6.2.1 Software Pipelining Information Terms

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.
- **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 functional units. 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 .LSD) = 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 4.6.3 for more information.

4.6.2.2 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_pipeline 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 4.6.3.2.
- **Cannot identify trip counter**. The loop trip counter could not be identified or was used incorrectly in the loop body.

4.6.2.3 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 six predicate registers. There are 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.
- **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.
- **Unsafe schedule for irregular loop**. "Irregular" loops are non-downcounting loops with a known number of iterations, such as a while loop. Irregular loops may require transformations that execute instructions more times than called for by the loop. This error means the compiler was unable to find a schedule with instructions that are safe to over-execute, are guarded with a predicate, or have their effects undone after the loop. Try to rewrite the loop as a down-counting loop. You may also try increasing the --speculate_loads (-mh) option.

4.6.2.4 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.

4.6.3 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=2 or --opt_for_space=3 options are invoked, the compiler increasingly favors code size over performance.

4.6.3.1 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 of 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 of the prolog and epilog 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
```

4.6.3.2 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.

The compiler and linker can provide automatic load speculation via the *auto* argument to the --speculate_loads option (i.e. *--speculate_loads=auto* or *-mh=auto*). Use of the *auto* argument makes it easier to use and benefit from speculative load optimizations. This option can generate speculative loads of up to 256 bytes beyond memory that the compiler can prove to be allocated.

In addition, the compiler communicates information to the linker to help automatically ensure the required pre- and post-padding:

- If the symbol of the speculatively loaded buffer is known at compile time, the linker ensures the object pointed to by the symbol has the required padding to let the speculative load access legal memory.
- If the symbol information is not known during compile time, the linker will ensure that the placement of data sections will allow legal accessing beyond the boundaries of the data sections. The linker does this by simply padding the start and end of the memory range(s) where the data sections are placed.

However, you can also specify the speculative loads threshold explicitly via the *--speculate_loads=n* option, where *n* is at least the minimum required pad (as explained earlier), but you also 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. If you choose the *auto* argument to --speculate_loads, the compiler will consider the larger threshold value automatically.

4.7 Redundant Loops

A loop iterates some number of times before the loop terminates. The number of iterations is called the *trip count*. The variable that counts iterations is the *trip counter*. When the trip counter reaches a limit equal to the trip count, the loop terminates. The Code Generation 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 set by the number of iterations executing in parallel. In Figure 4-1, 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 Code Generation 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 safe 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 4.4) or the use of the MUST_ITERATE pragma (see Section 7.9.22).

Note

Turning Off Redundant Loops: Specifying any --opt_for_space option turns off redundant loops.

4.8 Utilizing the Loop Buffer Using SPLOOP

The loop buffer 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 *TMS320C64x/C64x+ 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.

4.9 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.

Note

Disabling Code-Size Optimizations or Reducing the Optimization Level

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

Note

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

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

4.10 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.

4.10.1 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.

4.10.1.1 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 4.10.2) for consumption by phase 2 of feedback directed optimization.

4.10.1.2 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.

4.10.1.3 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. This option applies to all the C/C++ source files being compiled on the command-line.
If the environment variable TI_PROFDATA on the host is set, the data is written into the specified file. 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 to re-direct the profile data to a custom device driver routine. For example, this could be used to send the profile data to a device that does not use a file system.
Feedback directed optimization requires you to turn on at least some 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.

4.10.1.4 Example Use of Feedback Directed Optimization

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

1. Generate profile information.

```
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 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.

```
pdd6x -e foo.out -o pprofout.prf pprofout.pdat
```

4. Re-compile using the profile feedback file.

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

4.10.1.5 The .ppdata Section

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 has directives to place the .ppdata section in data memory. If the link command file has no section directive to allocate the .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.

4.10.1.6 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-grained, since it is selecting an option set for the whole function. Feedback directed optimization selects different optimization goals along specific regions within a function.

4.10.1.7 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 option 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.

4.10.1.8 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).
- Source code was modified since gen-profile. In this case, profile information is invalid for 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.

4.10.2 Profile Data Decoder

The code generation tools include a 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 4.10.1 for a discussion of 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.

Note

Profile Data Decoder Requirements: Compile applications with at least 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 to generate 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.

4.10.3 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.

4.10.4 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.
--analyze=codecov	Generates a code coverage information file and continues with profile-based compilation. Must be used with --use_profile_info.
--analyze_only	Generates only a code coverage information file. Must be used with --use_profile_info. Specify both --analyze=codecov and --analyze_only to do code coverage analysis of the instrumented application.

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

4.11 Using Profile Information to Get Better Program Cache Layout and Analyze Code Coverage

There are two different types of analysis information you can get from the path profiler: code coverage information and call graph information.

The program cache layout tool helps you to develop better program instruction cache efficiency into your applications. Program cache layout is the process of controlling the relative placement of code sections into memory to minimize the occurrence of conflict misses in the program instruction cache.

4.11.1 Background and Motivation

Effective utilization of the program instruction cache is an important part of getting the best performance from a C6000. The dedicated program instruction cache (L1P) provides fast instruction fetches, but a *cache miss* can be very costly. Some applications (e.g. h264) can spend 30%+ of the processor's time stalling due to L1P cache misses. A cache miss occurs when a fetch fails to read an instruction from L1P and the process is required to access the instruction from the next level of memory. A request to L2 or external memory has a much higher latency than an access from L1P.

Careful placement of code sections can greatly reduce the number of cache misses. The C6000 L1P is especially sensitive to code placement because it is *direct-mapped*.

Many L1P cache misses are *conflict misses*. Conflict misses occur when the cache has recently evicted a block of code that is then needed again. In a program instruction cache this often occurs when two frequently executed blocks of code (usually from different functions) interleave their execution and are mapped to the same cache line.

For example, suppose there is a call to function B from inside a loop in function A. Suppose also that the code for function A's loop is mapped to the same cache line as a block of code from function B that is executed every time that B is called. Each time B is called from within this loop, the loop code in function A is evicted from the cache by the code in B that is mapped to the same cache line. Even worse, when B returns to A, the loop code in A evicts the code from function B that is mapped to the same cache line.

Every iteration through the loop will cause two program instruction cache conflict misses. If the loop is heavily traversed, then the number of processor cycles lost to program instruction cache stalls can become quite large.

Many program instruction cache conflict misses can be avoided with more intelligent placement of functions that are active at the same time. Program instruction cache efficiency can be significantly improved using code placement strategies that utilize dynamic profile information that is gathered during the run of an instrumented application.

The program cache layout tool (clt6x) takes dynamic profile information in the form of a weighted call graph and creates a preferred function order command file that can be used as input to the linker to guide the placement of function subsections.

You can use the program cache layout tool to help improve your program locality and reduce the number of L1P cache conflict misses that occur during the run of your application, thereby improving your application's performance.

4.11.2 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.

4.11.2.1 Phase1 -- Collect Program Profile Information

In this phase, the compiler is invoked with --gen_profile_info, which instructs the compiler to add instrumentation code to collect profile information. The compiler inserts a small 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 4.10.2) for consumption by phase 2 of feedback directed optimization.

4.11.2.2 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.

4.11.3 What Performance Improvements Can You Expect to See?

If your application does not suffer from inefficient usage of the L1P cache, then the program cache layout capability will not have any effect on the performance of your application. Before applying the program cache layout tooling to your application, analyze the L1P cache performance in your application.

4.11.3.1 Evaluating L1P Cache Performance

Evaluating the L1P cache usage efficiency of your application will not only help you determine whether or not your application might benefit from using program cache layout, but it also gives you a rough estimate as to how much performance improvement you can reasonably expect from applying program cache layout.

There are several resources available to help you evaluate L1P cache usage in your application. One way of doing this is to use the Function Profiling capability in Code Composer Studio (CCS).

The number of CPU stall cycles that occur due to L1P cache misses gives you a reasonable upper bound estimate of the number of CPU cycles that you may be able to recover with the use of the program cache layout tooling in your application. Please be aware that the performance impact due to program cache layout will tend to vary for the different data sets that are run through your application.

4.11.4 Program Cache Layout Related Features and Capabilities

The code generation tools provide some features and capabilities that can be used in conjunction with the program cache layout tool, clt6x. The following is a summary:

4.11.4.1 Path Profiler

The code generation tools include a path profiling utility, pprof6x, that is run from the compiler, cl6x. The pprof6x utility is invoked by the compiler when the --gen_profile or the --use_profile command is used from the compiler command line:

cl6x --gen_profile ... *file.c*
cl6x --use_profile ... *file.c*

For further information about profile-based optimization and a more detailed description of the profiling infrastructure, see Section 4.10.

4.11.4.2 Analysis Options

The path profiling utility, pprof6x, appends code coverage or weighted call graph analysis information to existing CSV (comma separated values) files that contain the same type of analysis information.

The utility checks to make sure that an existing CSV file contains analysis information that is consistent with the type of analysis information it is being asked to generate (whether it be code coverage or weighted call graph analysis). Attempts to mix code coverage and weighted call graph analysis information in the same output CSV file will be detected, and pprof6x will emit a fatal error and abort.

--analyze=callgraph	Instructs the compiler to generate weighted call graph analysis information.
--analyze=codecov	Instructs the compiler to generate code coverage analysis information. This option replaces the previous --codecov option.
--analyze_only	Halts compilation after generation of analysis information is completed.

4.11.4.3 Environment Variables

To assist with the management of output CSV analysis files, pprof6x supports these environment variables:

TI_WCGDATA	Allows you to specify a single output CSV file for all weighted call graph analysis information. New information is appended to the CSV file identified by this environment variable, if the file already exists.
TI_ANALYSIS_DIR	Specifies the directory in which the output analysis file will be generated. The same environment variable can be used for both code coverage information and weighted call graph information (all analysis files generated by pprof6x will be written to the directory specified by the TI_ANALYSIS_DIR environment variable).

Note

TI_COVDIR Environment Variable

The existing TI_COVDIR environment variable is still supported when generating code coverage analysis, but is overridden in the presence of a defined TI_ANALYSIS_DIR environment variable.

4.11.4.4 Program Cache Layout Tool, clt6x

The program cache layout tool creates a preferred function order command file from input weighted call graph (WCG) information. The syntax is:

clt6x *CSV files with WCG info* **-o forder.cmd**

4.11.4.5 Linker

The compiler prioritizes the placement of a function relative to others based on the order in which --preferred_order options are encountered during the linker invocation. The syntax is:

--preferred_order= *function specification*

4.11.4.6 Linker Command File Operator unordered()

The new linker command file keyword unordered relaxes placement constraints placed on an output section whose specification includes an explicit list of which input sections are contained in the output section. The syntax is:

unordered()

4.11.5 Program Instruction Cache Layout Development Flow

Once you have determined that your application is experiencing some inefficiencies in its usage of the program instruction cache, you may decide to include the program cache layout tooling in your development to attempt to recover some of the CPU cycles that are being lost to stalls due to program instruction cache conflict misses.

4.11.5.1 Gather Dynamic Profile Information

The program cache layout tool, clt6x, relies on the availability of dynamic profile information in the form of a weighted call graph in order to produce a preferred function order command file that can be used to guide function placement at link-time when your application is re-built.

There are several ways in which this dynamic profile information can be collected. For example, if you are running your application on hardware, you may have the capability to collect a PC discontinuity trace. The discontinuity trace can then be post-processed to construct weighted call graph input information for the clt6x.

The method for collecting dynamic profile information that is presented here relies on the path profiling capabilities in the C6000 code generation tools. Here is how it works:

1. Build an instrumented application using the --gen_profile_info option.

   Using --gen_profile_info instructs the compiler to embed counters into the code along the execution paths of each function.

   To compile only use:

   cl6x *options* **--gen_profile_info** *src_file(s)*

   The compile and link use:

   cl6x *options* **--gen_profile_info** *src_file(s)* **-run_linker --library** *lnk.cmd*

2. Run an instrumented application to generate a .pdat file.

   When the application runs, the counters embedded into the application by --gen_profile_info keep track of how many times a particular execution path through a function is traversed. The data collected in these counters is written out to a profile data file named pprofout.pdat.

   The profile data file is automatically generated.
3. Decode the profile data file.

   Once you have a profile data file, the file is decoded by the profile data decoder tool, pdd6x, as follows:

   pdd6x -e= *instrumented app out file* **-o=pprofout.prf pprofout.pdat**

   Using pdd6x produces a .prf file which is then fed into the re-compile of the application that uses the profile information to generate weighted call graph input data.

4. Use decoded profile information to generate weighted call graph input.

   The --analyze compiler option tells the compiler to generate weighted call graph or code coverage analysis information. Its syntax is as follows:

--analyze=callgraph	Instructs the compiler to generate weighted call graph information.
--analyze=codecov	Instructs the compiler to generate code coverage information. This option replaces the previous --codecov option.

   The compiler also supports a new --analyze_only option which instructs the compiler to halt compilation after the generation of analysis information has been completed. This option replaces the previous --onlycodecov option.

   To make use of the dynamic profile information that you gathered, re-compile the source code for your application using the --analyze=callgraph option in combination with the --use_profile_info option:

   cl6x *options* **-mo --analyze=callgraph --use_profile_info=pprofout.prf** *src_file(s)*

   The use of -mo instructs the compiler to generate code for each function into its own subsection. This option provides the linker with the means to directly control the placement of the code for a given function.

   The compiler generates a CSV file containing weighted call graph information for each source file that is specified on the command line. If such a CSV file already exists, then new call graph analysis information will be appended to the existing CSV file. These CSV files are then input to the cache layout tool (clt6x) to produce a preferred function order command file for your application.

   For more details on the content of the CSV files (containing weighted call graph information) generated by the compiler, see Section 4.11.6.

4.11.5.2 Generate Preferred Function Order from Dynamic Profile Information

At this point, the compiler has generated a CSV file for each C/C++ source file specified on the command line of the re-compile of the application. Each CSV file contains weighted call graph information about all of the call sites in each function defined in the C/C++ source file.

The program cache layout tool, clt6x, collects all of the weighted call graph information in these CSV files into a single, merged weighted call graph. The weighted call graph is processed to produce a preferred function order command file that is fed into the linker to guide the placement of the functions defined in your application source files. This is the syntax for clt6x:

clt6x *.csv -o *forder.cmd*

The output of clt6x is a text file containing a sequence of **--preferred_order=** *function specification* options. By default, the name of the output file is forder.cmd, but you can specify your own file name with the -o option. The order in which functions appear in this file is their preferred function order as determined by the clt6x.

In general, the proximity of one function to another in the preferred function order list is a reflection of how often the two functions call each other. If two functions are very close to each other in the list, then the linker interprets this as a suggestion that the two functions should be placed very near to one another. Functions that are placed close together are less likely to create a cache conflict miss at run time when both functions are active at the same time. The overall effect should be an improvement in program instruction cache efficiency and performance.

4.11.5.3 Utilize Preferred Function Order in Re-Build of Application

Finally, the preferred function order command file that is produced by the clt6x is fed into the linker during the re-build of the application, as follows:

```
cl6x options --run_linker *.obj forder.cmd -l lnk.cmd
```

The preferred function order command file, forder.cmd, contains a list of --preferred_order=*function specification* options. The linker prioritizes the placement of functions relative to each other in the order that the --preferred_order options are encountered during the linker invocation.

Each --preferred_order option contains a function specification. A function specification can describe simply the name of the function for a global function, or it can provide the path name and source file name where the function is defined. A function specification that contains path and file name information is used to distinguish one static function from another that has the same function name.

The --preferred_order options are interpreted by the linker as suggestions to guide the placement of functions relative to each other. They are not explicit placement instructions. If an object file or input section is explicitly mentioned in a linker command file SECTIONS directive, then the placement instruction specified in the linker command file takes precedence over any suggestion from a --preferred_order option that is associated with a function that is defined in that object file or input section.

This precedence can be relaxed by applying the unordered() operator to an output specification as described in Section 4.11.7.

4.11.6 Comma-Separated Values (CSV) Files with Weighted Call Graph (WCG) Information

The format of the CSV files generated by the compiler under the **--analyze=callgraph --use_profile_info** option combination is as follows:

```
"caller","callee","weight" [CR][LF]
caller spec,callee spec,call frequency [CR][LF]
caller spec,callee spec,call frequency [CR][LF]
caller spec,callee spec,call frequency [CR][LF]
. . .
```

Keep the following points in mind:

- Line 1 of the CSV file is the header line. It specifies the meaning of each field in each line of the remainder of the CSV file. In the case of CSV files that contain weighted call graph information, each line will have a caller function specification, followed by a callee function specification, followed by an unsigned integer that provides the number of times a call was executed during run time.
- There may be instances where the caller and callee function specifications are identical on multiple lines in the CSV file. This will happen when a caller function has multiple call sites to the callee function. In the merged weighted call graph that is created by the clt6x, the weights of each line that has the same caller and callee function specifications will be added together.
- The CSV file that is generated by the compiler using the path profiling instrumentation will not include information about indirect function calls or calls to runtime support helper functions (like _remi or _divi). However, you may be able to gather information about such calls with another method (like the PC discontinuity trace mentioned earlier).
- The format of these CSV files is in compliance with the RFC-4180 specification of Comma-Separated Values (CSV) files. For more details on this specification, please see http://tools.ietf.org/html/rfc4180.

4.11.7 Linker Command File Operator - unordered()

The unordered() operator can be used in a linker command file. The effect of this operator is to relax the placement constraints placed on an output section specification in which the content of the output section is explicitly stated.

Consider an example output section specification:

```
SECTIONS
{
  .text:
  {
    file.obj(.text:func_a)
    file.obj(.text:func_b)
    file.obj(.text:func_c)
    file.obj(.text:func_d)
    file.obj(.text:func_e)
    file.obj(.text:func_f)
    file.obj(.text:func_g)
    file.obj(.text:func_h)
    *(.text)
  } > PMEM
  ...
}
```

In this SECTIONS directive, the specification of .text explicitly dictates the order in which functions are laid out in the output section. Thus by default, the linker will layout func_a through func_h in exactly the order that they are specified, regardless of any other placement priority criteria (such as a preferred function order list that is enumerated by --preferred_order options).

The unordered() operator can be used to relax this constraint on the placement of the functions in the '.text' output section so that placement can be guided by other placement priority criteria.

The unordered() operator can be applied to an output section as in Example 4-2.

Example 4-2. Output Section for unordered() Operator

```
SECTIONS
{
  .text: unordered()
  {
    file.obj(.text:func_a)
    file.obj(.text:func_b)
    file.obj(.text:func_c)
    file.obj(.text:func_d)
    file.obj(.text:func_e)
    file.obj(.text:func_f)
    file.obj(.text:func_g)
    file.obj(.text:func_h)
    *(.text)
  } > PMEM
  ...
}
```

So that, given this list of --preferred_order options:

- --preferred_order="func_g"
- --preferred_order="func_b"
- --preferred_order="func_d"
- --preferred_order="func_a"
- --preferred_order="func_c"
- --preferred_order="func_f"
- --preferred_order="func_h"
- --preferred_order="func_e"

The placement of the functions in the .text output section is guided by this preferred function order list. This placement will be reflected in a linker generated map file, as follows:

Example 4-3. Generated Linker Map File for Example 4-2

```
SECTION ALLOCATION MAP
output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.text       0    00000020    00000120
                   00000020    00000020     file.obj (.text:func_g:func_g)
                   00000040    00000020     file.obj (.text:func_b:func_b)
                   00000060    00000020     file.obj (.text:func_d:func_d)
                   00000080    00000020     file.obj (.text:func_a:func_a)
                   000000a0    00000020     file.obj (.text:func_c:func_c)
                   000000c0    00000020     file.obj (.text:func_f:func_f)
                   000000e0    00000020     file.obj (.text:func_h:func_h)
                   00000100    00000020     file.obj (.text:func_e:func_e)
```

4.11.7.1 About Dot (.) Expressions in the Presence of unordered()

Another aspect of the unordered() operator that should be taken into consideration is that even though the operator causes the linker to relax constraints imposed by the explicit specification of an output section's contents, the unordered() operator will still respect the position of a dot (.) expression within such a specification.

Consider the output section specification in Example 4-4.

Example 4-4. Respecting Position of a . Expression

```
SECTIONS
{
  .text: unordered()
  {
    file.obj(.text:func_a)
    file.obj(.text:func_b)
    file.obj(.text:func_c)
    file.obj(.text:func_d)
    . += 0x100;
    file.obj(.text:func_e)
    file.obj(.text:func_f)
    file.obj(.text:func_g)
    file.obj(.text:func_h)
    *(.text)
  } > PMEM
  ...
}
```

In Example 4-4, a dot (.) expression, ". += 0x100;", separates the explicit specification of two groups of functions in the output section. In this case, the linker will honor the specified position of the dot (.) expression with respect to the functions on either side of the expression. That is, the unordered() operator will allow the preferred function order list to guide the placement of func_a through func_d relative to each other, but none of those functions will be placed after the hole that is created by the dot (.) expression. Likewise, the unordered() operator allows the preferred function order list to influence the placement of func_e through func_h relative to each other, but none of those functions will be placed before the hole that is created by the dot (.) expression.

4.11.7.2 GROUPs and UNIONs

The unordered() operator can only be applied to an output section. This includes members of a GROUP or UNION directive.

Example 4-5. Applying unordered() to GROUPs

```
SECTIONS
{
  GROUP
  {
    .grp1:  {
      file.obj(.grp1:func_a)
      file.obj(.grp1:func_b)
      file.obj(.grp1:func_c)
      file.obj(.grp1:func_d)
    } unordered()
    .grp2:  {
      file.obj(.grp2:func_e)
      file.obj(.grp2:func_f)
      file.obj(.grp2:func_g)
      file.obj(.grp2:func_h)
    }
    .text:  { *(.text) }
  } > PMEM
  ...
}
```

The SECTIONS directive in Example 4-5 applies the unordered() operator to the first member of the GROUP. The .grp1 output section layout can then be influenced by other placement priority criteria (like the preferred function order list), whereas the .grp2 output section will be laid out as explicitly specified.

The unordered() operator cannot be applied to an entire GROUP or UNION. Attempts to do so will result in a linker command file syntax error and the link will be aborted.

4.11.8 Things to be Aware of

There are some behavioral characteristics and limitations of the program cache layout development flow that you should bear in mind:

- **Generation of Path Profiling Data File (.pdat).** When running an application that has been instrumented to collect path-profiling data (using --gen_profile_info compiler option during build), the application will use functions in the run-time-support library to write out information to the path profiling data file (pprofout.pdat in above tutorial). If there is a path profiling data file already in existence when the application starts to run, then any new path profiling data generated will be appended to the existing file. To prevent combining path profiling data from separate runs of an application, you need to either rename the path profiling data file from the previous run of the application or remove it before running the application again.
- **Indirect Calls Not Recognized by Path Profiling Mechanisms.** When using available path profiling mechanisms to collect weighted call graph information from the path profiling data, pprof6x does not recognize indirect calls. An indirect call site will not be represented in the CSV output file that is generated by pprof6x. You can work around this limitation by introducing your own information about indirect call sites into the relevant CSV file(s). If you take this approach, please be sure to follow the format of the callgraph analysis CSV file ("caller", "callee","call frequency"). If you are able to get weighted call graph information from a PC trace into a callgraph analysis CSV, this limitation will no longer apply (as the PC trace can always identify the callee of an indirect call).
- **Multiple --preferred_order Options Associated With Single Function.** There may be cases in which you might want to input more than one preferred function order command file to the linker during the link of an application. For example, you may have developed or received a separate preferred function order command file for one or more of the object libraries that are used by your application. In such cases, it is possible that one function may be specified in multiple preferred function order command files. If this happens, the linker will honor only the first instance of the --preferred_order option in which the function is specified.

4.12 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.

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

The compiler, when invoked with optimization, assumes that if the address of a local variable is passed to a function, the function changes the local variable by writing through the pointer. This makes the local variable's address unavailable for use elsewhere after returning. For example, the called function cannot assign the local variable's address to a global variable or return the local variable's address.

If your code uses aliases in this way and uses optimization, you must use the --aliased_variables option. For example, suppose your code is similar to the following, in which the address of the local variable x is passed to the function f(), which aliases glob_ptr to that address and returns the address. If this example were to be compiled with optimization, the --aliased_variables option would be needed in order for the function f() to be able to successfully perform its actions.

```
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;
}
```

4.12.2 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 4.12.1. 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 an expression value 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.

4.12.3 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 Section 5.4.

4.13 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.

4.14 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.

4.15 Using Performance Advice to Optimize Your Code

You can use compiler generated Performance Advice to optimize your code. To get Performance Advice, compile with the following options:

--advice:performance	Instructs the compiler to emit advice to stdout (default).
--advice:performance_file	Instructs the compiler to emit advice into a file.
--advice:performance_dir	Instructs the compiler to emit advice into a file in a specific directory.

Note

If an Advice file is requested, but there is no advice, the advice file will not be created; rather the compiler prints a message to stdout :

```
"filename.c": advice #27004: No Performance Advice is generated.
```

Example 1: The following example sends output advice sent to stdout (the default):

```
cl6x -mv6400+ -o2 -k --advice:performance func.c
```

```
"func.c", line 10: advice #30006: Loop at line 8 cannot be scheduled efficiently
   as it contains a function call ("_init"). Try making "_init" an inline
   function.
"func.c", line 12: advice #30000: Loop at line 8 cannot be scheduled efficiently
   as it contains a function call ("_calculate"). Try to inline call or
   consider rewriting loop.
```

Note that Advice to prevent Software Pipeline Disqualification (such as that presented above) will also be printed in the .asm file. So, func.asm will contain :

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*      Disqualified loop: Loop contains a call
;*          Loop at line 8 cannot be scheduled efficiently as it contains a
;*          function call ("_init"). Try making "_init" an inline function.
;*      Disqualified loop: Loop contains non-pipelinable instructions
;*      Disqualified loop: Loop contains a call
;*          Loop at line 8 cannot be scheduled efficiently as it contains a
;*          function call ("_calculate"). Try to inline call or consider
;*          rewriting loop.
;*      Disqualified loop: Loop contains non-pipelinable instructions
;*----------------------------------------------------------------------------*
```

Example 2: The following example sends output advice to a file named `filename.advice`:

```
cl6x -mv6400+ --advice:performance --advice:performance_file=filename.advice func.c
```

```
;******************************************************************************
;* TMS320C6x C/C++ Codegen         Unix v7.5.0P12047 (a0322878 - Feb 16 2012) *
;* Date/Time created: Thu Feb 16 10:26:02 2012                                *
;*                                                                            *
;* Warning: This file is auto generated by the compiler and can be            *
;* overwritten during the next compile.                                       *
;*                                                                            *
;******************************************************************************
;* User Options: --silicon_version=6400+

"func.c": advice #27000: Detecting compilation without optimization. Use
   optimization option -o2 or higher.
```

Example 3: The following examples send output advice to a file named `myfile.adv` in the `mydir` directory using various options.

Using the --advice:performance_file and --advice:performance_dir options:

```
cl6x -mv6400+ -o2 -k --advice:performance_file=myfile.adv --advice:performance_dir=mydir basicloop.c
```

Using only the --advice:performance_file option to specify the full path name:

```
cl6x -mv6400+ -o2 -k --advice:performance_file=mydir/myfile.adv basicloop.c
```

If --advice_dir option and full pathname are specified together, the --advice:performance_dir option is ignored, and the advice is generated in the full pathname advice file. Also, note that directory "mydir" must already exist for an advice file to be created there.

4.15.1 Advice #27000

```
advice #27000: Detecting compilation without optimization. Use optimization option -o2 or higher.
```

Your compilation is being done without any optimization options (-o0 and above). This prevents the compiler from using its most powerful optimization techniques, since the -o (--opt_level) options are the foundations for most other optimizations. You could get substantially better performance using -o2 (or above) optimization. Optimization option -o2 is required for the software pipelining loop optimization, which is crucial to getting good performance.

The C/C++ compiler is able to perform various optimizations, but you need to specify optimization options on the command line so that these optimizations are performed. The easiest way to invoke optimization is to specify the --opt_level=n option on the compiler command line. You can use -On 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. See "Invoking Optimization" in Section 4.1 for more information on Optimization Options.

4.15.2 Advice #27001 Increase Optimization Level

```
advice #27001: Detecting compilation with low optimization level.
       Use optimization option -o2 or higher.
```

Your compilation uses low-level optimization options (-o1 and below), which prevents the compiler from using its most powerful optimization techniques.

The C/C++ compiler is able to perform various optimizations, but you can control the level of these optimizations. High-level optimizations are performed in the optimizer and low-level, target-specific optimizations occur in the code generator. You must use high-level optimizations to achieve optimal code. You can invoke optimization by specifying the --opt_level=n option on the compiler command line.

See "Invoking Optimization" in Section 4.1 for more information on Optimization Options. Also see information for Advice #27000 in Section 4.15.1.

4.15.3 Advice #27002 Do not turn off software pipelining

```
advice #27002: Detecting compilation with "-mu" which turns off
       software pipelining. To optimize, turn off this option.
```

Your compilation is being done using -mu, which turns off software-pipelining. Software-pipelining is a key optimization for achieving good performance. This Advice is issued to alert you to NOT use compiler option -mu. -mu is a good option for debugging, but it is recommended that this option not be used for production code because of the negative performance implications.

In general, to achieve maximal performance, avoid using the following in production code :

- -g: Compiling with debug information no longer affects the ability to optimize code. However, high levels of optimization do make it more difficult to debug code due to code restructuring and other transformations. If you are still at the debugging stage, you may want to use a lower level of optimization. For production code, you can use a high level of optimization with or without disabling the inclusion of debug information.
- -ss: Interlist source code into assembly file. As with -g, this option can negatively impact performance.
- -mu: Turns off software-pipelining, which is a key optimization for achieving good performance. This is a good option for debugging, but is not recommended for use in production code due to negative performance implications.

4.15.4 Advice #27003 Avoid compiling with debug options

```
advice #27003: Detecting compilation with debug option "-g",
       which hinders optimization. To optimize, remove -g or
    compile with --optimize_with_debug_option.
```

This advice was provided in earlier versions in which the inclusion of debug information impacted the ability to optimize code. Debug information no longer impacts optimization, and the --optimize_with_debug option has been deprecated. Also see Advice #27002 in Section 4.15.3.

4.15.5 Advice #27004 No Performance Advice generated

```
advice #27004: No Performance Advice is generated.
```

The compiler detects that your compilation is being done using --advice:performance option, but the compiler has no Advice to report. This Advice is issued to alert you to the fact that no Advice is being emitted, and an Advice file will not be created (if one was requested).

4.15.6 Advice #30000 Prevent Loop Disqualification due to call

```
advice #30000: Loop at line 10 cannot be scheduled efficiently,
         as it contains a function call ("function_name").
         Try to inline call or consider rewriting loop.
```

The compiler attempts to perform the software pipeline loop optimization at optimization level --opt_level=3 (or -O3). If there is a call in the loop, the compiler will attempt to completely inline the called function, but sometimes this is not possible. If the compiler cannot inline the called function, software pipelining cannot be performed. This can severely reduce the performance of the loop.

In the test case below, the call to the function "func2" prevents software pipelining. Inlining function "func2" or rewriting the loop to avoid a function call can avoid pipeline disqualification. If the loop pipelines successfully you may see performance improvement.

```
void func1(int *p, int *q, int n)
{
       unsigned int i;

       for (i = 0; i < n; i++)
       {
           int t = func2(i);

           p[i] = q[i] + t;
       }
}
```

4.15.7 Advice #30001 Prevent Loop Disqualification due to rts-call

```
advice #30001: Loop at line 18 cannot be scheduled efficiently, as it
                  contains conversion from "type-a" to "type-b".
                  Try to use "suggested" type.
```

The compiler can insert calls to special functions in the run-time support library (RTS) to support operations that are not natively supported by the ISA. For instance, while floating-point ISAs support instructions to convert between floating-point and signed integer values, they don't support conversion between floating-point and unsigned integer values. If you use unsigned variables in floating point expressions, the compiler will generate a call to an RTS routine to carry out this function. Such a call will disable software pipelining.

You can change the unsigned variables in your code to signed variables and prevent this from happening. The compiler will then be able to use the native hardware instead of adding the special function call, so you may get better performance.

4.15.8 Advice #30002 Prevent Loop Disqualification due to asm statement

```
advice #30002: Loop at line 8 cannot be scheduled efficiently, as it
                  contains an asm() statement. Try to replace the asm()
                  statement with C or intrinsic statement.
```

An asm statement inserted in a C code loop will disqualify the loop for software pipelining. Software-pipelining is a key optimization for achieving good performance. You may see reduced performance without software pipelining.

Replace the asm() statement with native C, or an intrinsic function call to prevent this from happening.

4.15.9 Advice #30003 Prevent Loop Disqualification due to complex condition

```
advice #30003: Loop at line 8 cannot be scheduled efficiently, as it
                  contains complex conditional expression. Try to simplify
                  condition.
```

Your code contains a complex conditional expression, possibly a large "if" clause, within a loop, which is preventing optimization. The compiler will optimize small “if” statements (“if” statements with “if” and “else” blocks that are short or empty). The compiler will not optimize large "if" statements, and such large if statements within the loop body will disqualify the loop for software pipelining. Software-pipelining is a key optimization; you may see reduced performance without it.

In the examples below, Example 1 will pipeline, but Example 2 won't :

Example 1:

```
for (i=0; i < N; i++)
{
        if (!flag)        {
           //statements
        }
        else              {
          x[i] = y[i];
        }
}
```

Example 2:

```
for (i = 0; i < n; i++)
{
        if (!flag)        {
           //statements
        }
        else              {
           if (flag == 1) x[i] = y[i];
        }
}
```

Example 1 will have significantly better performance than Example 2 because it pipelines successfully. But Example 2 can be pipelined if the code is modified to eliminate the nested "if" :

```
for (i = 0; i < n; i++)
{
        if (!flag)        {
           //statements
        }
        else              {
           p = (flag == 1);
           x[i] = !p * x[i] + p * y[i] ;
        }
}
```

4.15.10 Advice #30004 Prevent Loop Disqualification due to switch statement

```
advice #30004: Loop at line 257 cannot be scheduled efficiently, as it
                   contains a switch statement. Try to rewrite loop.
```

There is a switch statement within the loop. A switch statement in a loop will disqualify the loop for software pipelining. Software-pipelining is a key optimization; you may see reduced performance without it.

Try and rewrite the loop without a switch statement.

4.15.11 Advice #30005 Prevent Loop Disqualification due to arithmetic operation

```
advice #30005: Loop at line 5 cannot be scheduled efficiently, as it
               contains a "division" operation. Rewrite using simpler
               operations if possible.
```

The compiler can insert calls to special functions in the run-time support library (RTS) to support operations that are not natively supported by the ISA. For example, the compiler calls __c6xabi_divi() function to perform 32-bit integer divide operation. Such functions are called compiler helper functions, and result in a function call within the loop body. In the example below, the compiler will accomplish the division operation by calling the compiler helper function "_divi" :

```
void func(float *p, float n)
{
        int i;

        for (i = 1; i < 1000; i++)    {
            p[i] /= n;
        }
}
```

However if we modify this loop, like below, the loop pipelines :

```
void func_adjusted(float *p, float n)
{
        int i;

        float inv = 1/n;

        for (i = 1; i < 1000; i++)     {
            p[i] *= inv;
        }
}
```

4.15.12 Advice #30006 Prevent Loop Disqualification due to call(2)

```
advice #30006: Loop at line 22 cannot be scheduled efficiently, as it
                  contains a function call ("function_name"). Try making "function_name" an
                  inline function.
```

For improved performance, at optimization levels --opt_level=2 (-O2) and --opt_level=3 (-O3), the compiler attempts to software pipeline your loops. 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, and the loop could not be efficiently scheduled.

For example, in the test case below, call to function "func2" prevents software pipelining:

```
void func1(int *p, int *q, int n)
{
       unsigned int i;

       for (i = 0; i < n; i++)       {
           int t = func2(i);

           ; other operations
       }
}
int function func2() { . . . }
```

However if function func2 is inlined, it saves the overhead of a function call. The compiler is free to optimize the function in context with surrounding code. Automatic inlining is controlled by the "inline" keyword; use it to allow inlining of specific functions :

```
inline int function func2() { . . . }
```

Also see #Advice 30000 in Section 4.15.6.

4.15.13 Advice #30007 Prevent Loop Disqualification due to rts-call(2)

```
advice #30007: Attempting to use floating-point operation "__mpyd" on
                  fixed-point device, at line 5 (there may be other instances
                  of this). Such calls reduce loop performance; use fixed point
                  operation if possible.
```

The compiler inserts calls to special functions in the run-time support library (RTS) to support operations that are not natively supported by the instruction set architecture (ISA). For example, fixed point ISAs do not support floating-point instructions and the compiler will generate a call to an RTS routine to carry out the floating point operation. In the test case below, the floating-point multiplication is unavailable for a fixed-point device :

```
void func(float *p, float *q, int n)
{
        unsigned int i;

        for (i = 1; i < n; i++)
        {
            p[i] = (q[i] * 12.4) / p[i - 1];
        }
}
```

If compiled for C6400+ (compiler option -mv6400+) the compiler will use an RTS call to carry out the operation. Such a call will disable software pipelining. You can rewrite the operation, or use a fixed point operation to prevent this.

Also see Advice #30001 in Section 4.15.7.

4.15.14 Advice #30008 Improve Loop; Qualify with restrict

```
advice #30008: Consider adding the restrict qualifier to the definition
                  of inp1, inp2 if they don't access the same memory location.
```

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.

To see more information on using restrict, refer to Section 7.5.6

4.15.15 Advice #30009 Improve Loop; Add MUST_ITERATE pragma

```
advice #30009: If you know that this loop will always execute at a
                  multiple of <2> and at least <2> times, try adding
                  "#pragma MUST_ITERATE(2, ,2)" just before the loop.
```

The C6000 architecture is partitioned into two nearly symmetric halves. The resource breakdown displayed in the software pipelining information in the asm file, is computed after the compiler has partitioned instructions to either the A-side or the B-side. If the resources are imbalanced (i.e.; some resources on one side are used more than resources on the other) software pipelining is resource-bound, and the loop cannot be efficiently scheduled. If the compiler has information about the trip-count for the loop, it can unroll the loop to balance resource usage, and get better pipelining. You can give loop trip-count information to the compiler using the "MUST_ITERATE" pragma.

To see more information on using the MUST_ITERATE pragma, refer to Section 7.9.22

4.15.16 Advice #30010 Improve Loop; Add MUST_ITERATE pragma(2)

```
advice #30010: If you know that this loop will always execute at least
                  <2> times, try adding "#pragma MUST_ITERATE(2)" just before the loop.
```

See Advice #30009 at Section 4.15.15.

4.15.17 Advice #30011 Improve Loop; Add _nasssert()

```
advice #30011: Consider adding assertions to indicate n-byte alignment
                  of variables input1, input2, output if they are actually n-byte
                  aligned: _nassert((int)(input1) % 8 == 0).
```

Most loops have memory access instructions. The compiler attempts to use wider load instructions, and aligned memory accesses instead of non-aligned memory accesses to reduce/balance out resources used for the memory access instructions. One of the ways to let the compiler know that it is safe to use "wider" loads is to use the keyword "_nassert".

To find out more on using the _nassert keyword, see Section 8.6.11.

4.16 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.

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.

For example, suppose the following C code is compiled with optimization (--opt_level=2) and --optimizer_interlist options:

```
int copy (char *str, const char *s, int n)
{
   int i;
   for (i = 0; i < n; i ++)
      *str++ = *s++;
}
```

The assembly file contains compiler comments interlisted with assembly code.

```
_main:
;** 5-----------------------    printf("Hello, world\n");
;** 6-----------------------    return 0;
           STW     .D2     B3,*SP--(12)
.line3
           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
.line4
           ZERO    .L1     A4
.line5
           LDW     .D2     *++SP(12),B3
           NOP             4
           B       .S2     B3
           NOP             5
           ; BRANCH OCCURS
```

If you add the --c_src_interlist option (compile with --opt_level=2, --c_src_interlist, and --optimizer_interlist), the assembly file contains compiler comments and C source interlisted with assembly code.

```
_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
```

4.17 Debugging and Profiling Optimized Code

The compiler generates symbolic debugging information by default at all optimization levels. Generating debug information does not affect compiler optimization or generated code. However, higher levels of optimization negatively impact the debugging experience due to the code transformations that are done. For the best debugging experience use --opt_level=off.

The default optimization level is off.

Debug information increases the size of object files, but it does not affect the size of code or data on the target. If object file size is a concern and debugging is not needed, use --symdebug:none to disable the generation of debug information.

If you are having trouble debugging loops in your code, you can use the --disable_software_pipeline option to turn off software pipelining. See Section 4.6.1 for more information.

4.17.1 Profiling Optimized Code

To profile optimized code, use optimization (--opt_level=0 through --opt_level=3).

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.

4.18 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. The following are some of the optimizations performed by the compiler:

Optimization	See
Cost-based register allocation	Section 4.18.1
Alias disambiguation	Section 4.18.2
Branch optimizations and control-flow simplification	Section 4.18.3
Data flow optimizations • Copy propagation • Common subexpression elimination • Redundant assignment elimination	Section 4.18.4
Expression simplification	Section 4.18.5
Inline expansion of functions	Section 4.18.6
Function symbol aliasing	Section 4.18.7
Induction variables and strength reduction	Section 4.18.8
Loop-invariant code motion	Section 4.18.9
Loop rotation	Section 4.18.10
Vectorization	Section 4.18.11
Instruction scheduling	Section 4.18.12

C6000-Specific Optimization	See
Register variables	Section 4.18.13
Register tracking/targeting	Section 4.18.14
Software pipelining	Section 4.18.15

4.18.1 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.

4.18.2 Alias Disambiguation

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

4.18.3 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.

This type of optimization is enabled by the --opt_level=0 and higher optimization settings.

4.18.4 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. This type of optimization is enabled by the --opt_level=1 and higher optimization settings.
- **Common subexpression elimination.** When two or more expressions produce the same value, the compiler computes the value once, saves it, and reuses it. This type of optimization is enabled by the --opt_level=2 and higher optimization settings.
- **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. This type of optimization is enabled by the --opt_level=1 for local assignments and --opt_level=2 for global assignments.

4.18.5 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.

This type of optimization is enabled by the --opt_level=0 and higher optimization settings.

4.18.6 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. For information about interactions between command-line options, pragmas, and keywords that affect inlining, see Section 3.11.

This type of optimization is enabled by the --opt_level=0 and higher optimization settings.

4.18.7 Function Symbol Aliasing

The compiler recognizes a function whose definition contains only a call to another function. If the two functions have the same signature (same return value and same number of parameters with the same type, in the same order), then the compiler can make the calling function an alias of the called function.

For example, consider the following:

```
int bbb(int arg1, char *arg2);
int aaa(int n, char *str)
{
   return bbb(n, str);
}
```

For this example, the compiler makes aaa an alias of bbb, so that at link time all calls to function aaa should be redirected to bbb. If the linker can successfully redirect all references to aaa, then the body of function aaa can be removed and the symbol aaa is defined at the same address as bbb.

For information about using the GCC function attribute syntax to declare function aliases, see Section 7.14.2

4.18.8 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.

This type of optimization is enabled by the --opt_level=2 and higher optimization settings.

4.18.9 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.

4.18.10 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.

This type of optimization is enabled by the --opt_level=0 and higher optimization settings.

4.18.11 Vectorization (SIMD)

The compiler may convert a loop such that it uses instructions that operate on more than one piece of data at a time, increasing the performance dramatically.

4.18.12 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.

4.18.13 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.

This type of optimization is enabled by the --opt_level=0 and higher optimization settings.

4.18.14 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.

4.18.15 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 4.6 for more information.

This type of optimization is enabled by the --opt_level=2 and higher optimization settings.

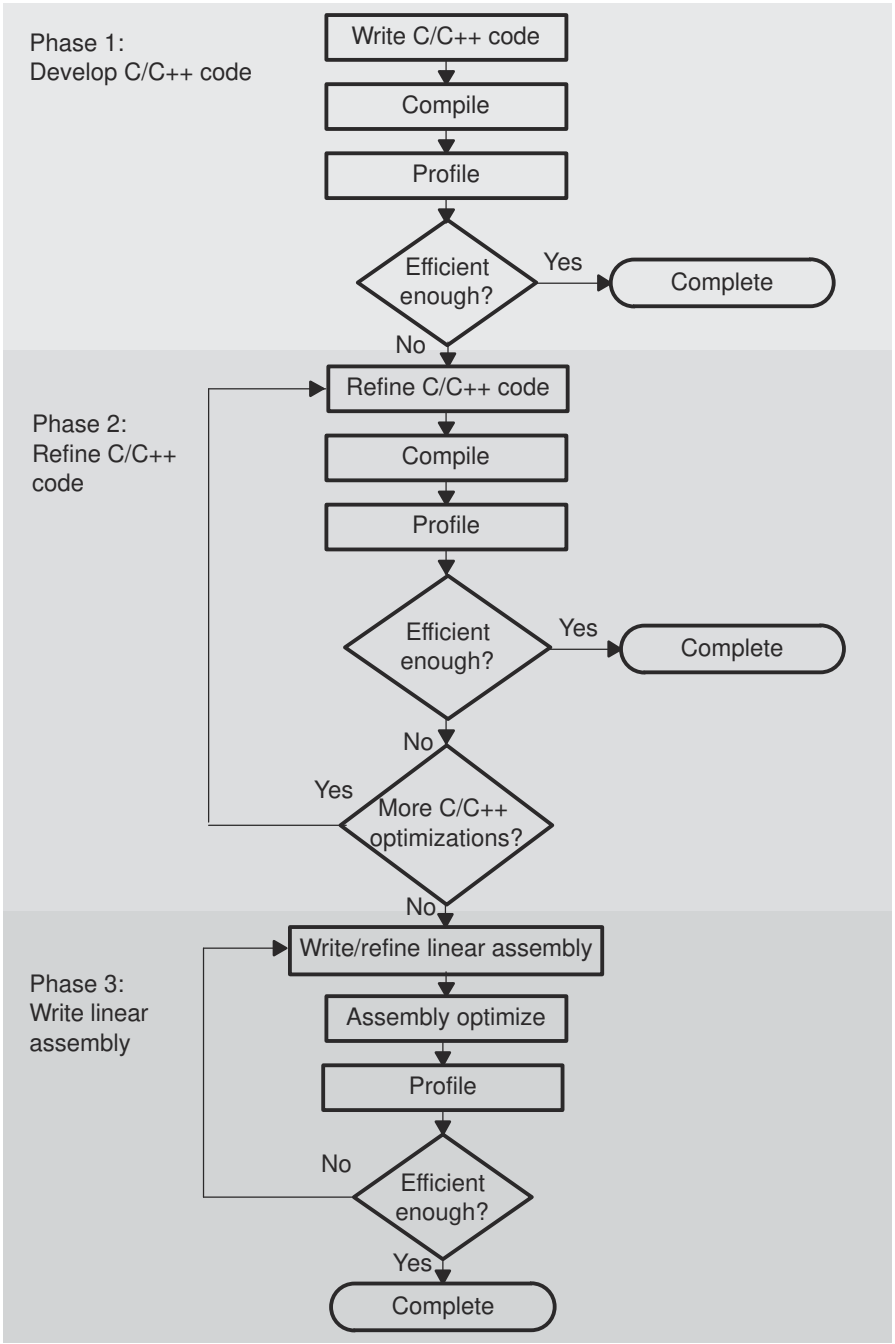
Chapter 5

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.

5.1 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. Compile with the --opt_level=3 option and without any --debug option. Identify any inefficient areas in your C/C++ code. See Section 4.17 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. 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.

5.2 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 4.6.

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 3.

5.3 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 5.4.
- **Options that affect what the assembly optimizer does**

  The compiler options in Table 5-1 affect the behavior of the assembly optimizer.

Table 5-1. Options That Affect the Assembly Optimizer

Option	Effect	See
--ap_extension	Changes the default extension for assembly optimizer source files	Section 3.3.10
--ap_file	Changes how assembly optimizer source files are identified	Section 3.3.8
--disable_software_pipelining	Turns off software pipelining	Section 4.6.1
--debug_software_pipeline	Generates verbose software pipelining information	Section 4.6.2
--interrupt_threshold=*n*	Specifies an interrupt threshold value	Section 3.12
--keep_asm	Keeps the assembly language (.asm) file	Section 3.3.2
--no_bad_aliases	Presumes no memory aliasing	Section 4.12.3
--opt_for_space=*n*	Controls code size on four levels (*n*=0, 1, 2, or 3)	Section 4.9
--opt_level=*n*	Increases level of optimization (*n*=0, 1, 2, or 3)	Section 4.1
--quiet	Suppresses progress messages	Section 3.3.2
--silicon_version=*n*	Select target version	Section 3.3.5
--skip_assembler	Compiles or assembly optimizes only (does not assemble)	Section 3.3.2
--speculate_loads=*n*	Allows speculative execution of loads with bounded address ranges	Section 4.6.3

- **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.

Note

Do Not Use Scheduled Assembly Code as Source

The assembly optimizer assumes that the instructions in the input file are placed in the logical order in which you would like them to occur (that is, linear assembly code). Parallel instructions are illegal.

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 5.3.1 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 5.3.2 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 5.3.4, for an illustration of the use of source comments and the resulting assembly optimizer output.

5.3.1 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, 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.

5.3.2 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 using two directives. The **.rega** directive constrains a symbolic name to A-side registers. The **.regb** directive constrains 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 5-1 is a hand-coded linear assembly program that computes a dot product; compare to Example 5-2, which illustrates C code.

Example 5-1. 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 5-2 is refined C code for computing a dot product.

Example 5-2. 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
```

In the linear assembler, you can also specify register pairs using the .cproc and/or .reg directive as in Example 5-3:

Example 5-3. Specifying a Register Pair

```
   .global foopair
foopair: .cproc q1:q0,s0
   .reg r1:r0
   ADD q1:q0, s0, r1:r0
   .return r1:r0
   .endproc
```

In Example 5-3, the expression "q1:q0" means that the first argument into the linear assembly function is a register pair. By the C calling conventions, the pair "q1:q0" symbols are mapped to register pair "a5:a4". When a register pair syntax is used as the argument to a .reg directive (as shown), it means that the two register symbols are constrained to be an aligned register pair when the compiler processes the linear assembler source and allocates actual registers that the register pair symbols map to "r1:r0" as shown.

The compiler supports a register quad syntax (C6600 only) in order to specify 128-bit operands of 128-bit capable instructions in linear assembly and assembly source code. Example 5-4 illustrates how you can specify register quads:

Example 5-4. Specifying a Register Quad (C6600 Only)

```
    .global fooquad
fooquad: .cproc q3:q2:q1:q0, s3:s2:s1:s0
    .reg r3:r2:r1:r0
    QMPY32 s3:s2:s1:s0, q3:q2:q1:q0, r3:r2:r1:r0
    .return r3:r2:r1:r0
    .endproc
```

In Example 5-4, the expression "q3:q2:q1:q0" means that the first argument into the linear assembly function is a register quad. By the C calling conventions, the quad "q3:q2:q1:q0" symbols are mapped to register quad "a7:a6:a5:a4". When a register quad syntax is used as the argument to a .reg directive (as shown), it means that the four register symbols are constrained to be an aligned register quad when the compiler processes the linear assembler source and allocates actual registers that the register quad symbols map to "r3:r2:r1:r0" as shown.

5.3.3 Functional Unit Specification for Linear Assembly

Specifying functional units has been deprecated by the ability to partition registers directly. (See Section 5.3.2 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*.

5.3.4 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 5-5 shows code for a function called Lmac that contains comments.

Example 5-5. 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
```

5.3.5 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.

5.4 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 5-2 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 5-2 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 5-2. 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*$_1$ *I register*$_1$ [, *symbol*$_2$ *I 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*$_1$ *I register*$_1$ [, *symbol*$_2$ *I 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 I 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 8.3 and Section 8.4. 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 7.12 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
```

.call (continued)

Calls a Function

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*$_2$ l*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 8.3 and Section 8.4.

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 8.4). 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.

.cproc/.endproc (continued)

Define a C Callable Procedure

- **Register pairs**. A register pair is specified as arghi:arglo and represents a 40-bit argument or a 64-bit type double argument.

  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.
- **Register quads** (C6600 only). A register quad is specified as r3:r2:r1:r0 and represents a 128-bit type, __x128_t. See Example 5-4.

If you are calling a procedure from C++ source, you must use the appropriate linkname for the procedure label. Otherwise, you can force C naming conventions by using the extern C declaration. See Section 7.12 and Section 8.6 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 5.4.1 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 5.5.

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 5.4.1 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 (or register quads for C6600) explicitly is optional. Doing so is only necessary if the registers should be allocated as a pair, but they are not used that way. It is a best practice to declare register pairs and register quads with the pair/quad syntax. Here is an example of declaring a register pair:

```
   .regA7: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
```

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
```

Note

Reserving Registers A4 and A5

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

Example 1

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

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

Example 2

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

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

.return

Return a Value to a C callable Procedure

Syntax

.return [*argument*]

Description

The **.return** directive function is equivalent to the return statement in C/C++ code. It places the optional argument in the appropriate register for a return value as per the C/C++ calling conventions (see Section 8.4).

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 the loop can iterate. A *factor* of 2 states that your loop always executes an even number of times, allowing the compiler to unroll once; this can improve performance. In this example, the loop executes some multiple of 8, between 8 and 48, times: `loop:  .trip 8, 48, 8` 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 unpipelined loop is executed based on a comparison of 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 4.7.

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 3.12 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
```

5.4.1 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 cannot be written to. Instructions that write to B15 are not allowed in .proc or .cproc regions. Stack space can be allocated by the assembly optimizer in .proc or .cproc regions to store 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
```

  These illegal example .if/.endif loops are partly inside and partly outside .cproc or .proc regions:

```
       .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
  - ADDKSP and DP-relative addressing

5.5 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 5-1. 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.

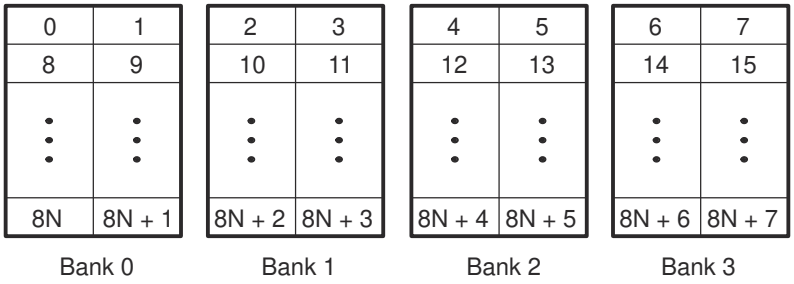


Figure 5-1. 4-Bank Interleaved Memory

For devices that have more than one memory space (Figure 5-2), 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.

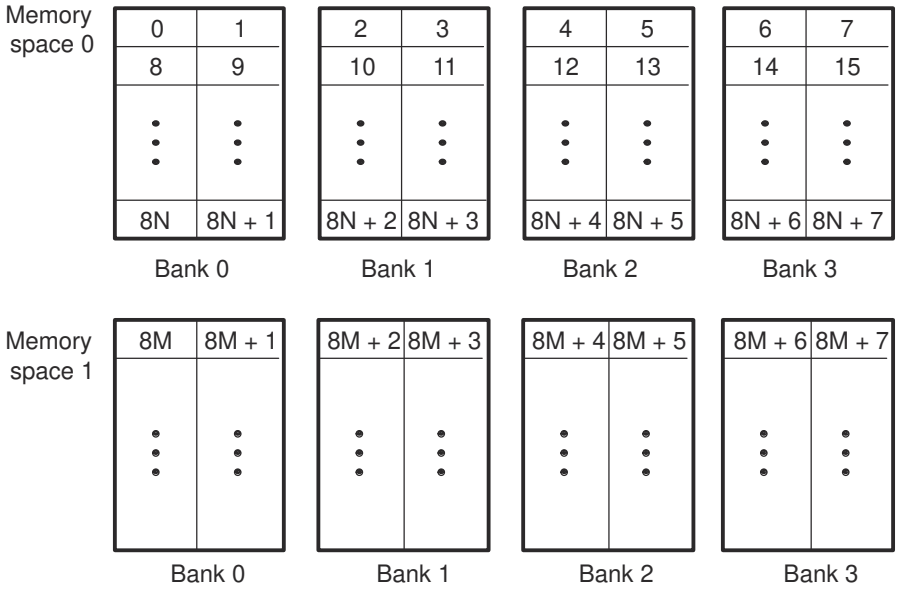


Figure 5-2. 4-Bank Interleaved Memory With Two Memory Spaces

5.5.1 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 . 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 5-6 shows loads and stores extracted from a loop that is being software pipelined.

Example 5-6. Load and Store Instructions That Specify Memory Bank Information

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

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

.mptr   Aout,optr+0,4
.mptr   Bout,optr+2,4
LDW          *Ain--[2],Ain12          ; IN(k-I) & IN(k-I+1)
LDW          *Bin--[2],Bin23          ; IN(k-I-2) & IN(k-I-1)
LDW          *Ain--[2],Ain34          ; IN(k-I-4) & IN(k-I-3)
LDW          *Bin--[2],Bin56          ; IN(k-I-6) & IN(k-I-5)

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

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

5.5.2 A Dot Product Example That Avoids Memory Bank Conflicts

The C code in Example 5-7 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 5-8 implement the dotp loop kernel. Example 5-9 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
```

  No bank conflict:

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

Example 5-7. 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 5-8. 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 5-9. 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
   || LDW   *B0, B1
```

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 5-10 shows the unrolled loop kernel.

Example 5-10. Dot Product From Example 5-8 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 5-11. Unrolled Dot Product Kernel From Example 5-9

```
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 5-10, 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 5-11 can never have a memory bank conflict.

In Example 5-8, 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.

5.5.3 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 5-12, the .mptr directive is used to specify which memory accesses conflict and which never conflict.

Example 5-12. 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]    ;
```

5.5.4 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
```

5.6 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.

5.6.1 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.

5.6.2 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 4.12.2.

5.6.3 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.

Note

Memory Dependency Exception

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.

5.6.4 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 5-13. 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...:
.mdep ld1, st1 <<bold>>
```

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 5-14 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 5-14. Software Pipeline Using .mdep ld1, st1

```
      iteration n             iteration n+1
      -------------           -------------
      LDW { ld1 }
      ...                     LDW { ld1 }
      STW { st1 }             ...
                              STW { st1 }
*<If that schedule...>
   .mdep   st1, ld1
```

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 5-15.

Example 5-15. 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,...>
    .mdep   ld1, st1
    .mdep   st1, ld1
```

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
```

5.6.5 Memory Alias Examples

Following are memory alias examples that use the .mdep and .no_mdep directives.

- **Example 1**

  The .mdep r1, r2 directive declares that LDW must be before STW. In this case, src and dst might point to the same array.

```
    fn:     .cproc      dst, src, cnt
            .reg        tmp
            .no_mdep
            .mdep       r1, r2
            LDW         *src{r1}, tmp
            STW         cnt, *dst{r2}
            .return     tmp
            .endproc
```

- **Example 2**

  Here, .mdep r2, r1 indicates that STW must occur before LDW. Since STW is after LDW in the code, the dependence relation is across loop iterations. The STW instruction writes a value that may be read by the LDW instruction on the next iteration. In this case, a 6-cycle recurrence is created.

```
    fn:     .cproc     dst, src, cnt
            .reg        tmp
            .no_mdep
            .mdep       r2, r1
    LOOP:   .trip       100
            LDW         *src++{r1}, tmp
            STW         tmp, *dst++{r2}
     [cnt]  SUB         cnt, 1, cnt
     [cnt]  B           LOOP
            .endproc
```

Note

Memory Dependence/Bank Conflict

Do not confuse 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.

Note

Volatile References

For volatile references, use .volatile rather than .mdep.

This page intentionally left blank.

Chapter 6

Linking C/C++ Code

The C/C++ Code Generation 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*.

6.1 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.

6.1.1 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 without listing any C/C++ files to be compiled on the command line, you *must* use **--rom_model** or **--ram_model** on the command line or in the linker command file. The --rom_model option uses automatic variable initialization at run time; the --ram_model option uses variable initialization at load time. See Section 6.3.4 for details about using the --rom_model and --ram_model options. If you fail to specify the ROM or RAM model, you will see a linker warning that says: `warning: no suitable entry-point found; setting to 0`
filenames	Names of object files, linker command files, or archive libraries. The default extensions for input files are .c.obj (for C source files) and .cpp.obj (for C++ source files). 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.
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 detail in the *TMS320C6000 Assembly Language Tools User's Guide*.)
--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.

Note

The default file extensions for object files created by the compiler have been changed. Object files generated from C source files have the .c.obj extension. Object files generated from C++ source files have the .cpp.obj extension.

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 object files prog1.c.obj, prog2.c.obj, and prog3.cpp.obj, with an executable object file filename of prog.out with the command:

```
cl6x --run_linker --ram_model prog1 prog2 prog3 --output_file=prog.out
     --library=rts6600.lib
```

6.1.2 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 6.1.1.

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 3.2.

You can compile and link a C/C++ program consisting of object files prog1.c, prog2.c, and prog3.c, with an executable object file filename of prog.out with the command:

```
cl6x prog1.c prog2.c prog3.c --run_linker --rom_model --output_file=prog.out --library=rts6600.lib
```

When you use cl6x --run_linker *after* listing at least one C/C++ file to be compiled on the same command line, by default the **--rom_model** is used for automatic variable initialization at run time. See Section 6.3.4 for details about using the --rom_model and --ram_model options.

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

6.1.3 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.

6.2 Linker Code Optimizations

6.2.1 Conditional Linking

With ELF conditional linking, a code or data section will not be included in a link unless at least one symbol in that code or data section is referenced.

You can use the RETAIN pragma (Section 7.9.30) to force the section that contains a specific symbol to be included in the link. You can use the CODE_SECTION (Section 7.9.3) and DATA_SECTION (Section 7.9.6) pragmas to force a symbol to be allocated in a particular section. The symbol must be referenced in a statement other than its declaration to force that section to be included in the link.

6.2.2 Generating Function Subsections (--gen_func_subsections Compiler Option)

The compiler translates a source module into an object file. It may place all of the functions into a single code section, or it may create multiple code sections. The benefit of multiple code sections is that the linker may omit unused functions from the executable.

When the linker collects code to be placed into an executable file, it cannot split code sections. If the compiler did not use multiple code sections, and any function in a particular module needs to be linked into the executable, then all functions in that module are linked in, even if they are not used.

An example is a library *.c.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. If only one code section was used, both the signed and unsigned routines are linked in since they exist in the same *.c.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 using the --gen_func_subsections option is to place each function in its own file.

If this option is not used, the default is "off". If this option is used but neither "on" nor "off" is specified, the default is "on".

6.2.3 Generating Aggregate Data Subsections (--gen_data_subsections Compiler Option)

Similarly to code sections described in the previous section, data can either be placed in a single section or multiple sections. The benefit of multiple data sections is that the linker may omit unused data structures from the executable. This option causes aggregate data—arrays, structs, and unions—to be placed in separate subsections of the data section.

If this option is not used, the default is "on". If this option is used but neither "on" nor "off" is specified, an error message is provided.

If the SET_DATA_SECTION pragma is used, the --gen_data_subsections=on option is ignored. User-defined section placement takes precedence over automatic generation of subsections.

6.3 Controlling the Linking Process

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

- Include the compiler's run-time-support library
- Specify the type of boot-time 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*.

6.3.1 Including the Run-Time-Support Library

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. The following sections describe two methods for including the run-time-support library.

6.3.1.1 Automatic Run-Time-Support Library Selection

The linker assumes you are using the C and C++ conventions if either the --rom_model or --ram_model linker option is specified, or if at least one C/C++ file to compile is listed on the command line. See Section 6.3.4 for details about using the --rom_model and --ram_model options.

If the linker assumes you are using the C and C++ conventions 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 most compatible run-time-support library for your program. The run-time-support library chosen by the compiler is searched after any other libraries specified with the --library option on the command line or in the linker command file. If libc.a is explicitly used, the appropriate run-time-support library is included in the search order where libc.a is specified.

You can disable the automatic selection of a run-time-support library by using 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 than the one reported by --issue_remarks, you must specify the name of the desired run-time-support library using the --library option and in your linker command files when necessary.

Example 6-1. Using the --issue_remarks Option

```
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"
```

6.3.1.2 Manual Run-Time-Support Library Selection

You can bypass automatic library selection by explicitly specifying the desired run-time-support library to use. Use the --library linker option to specify the name of the library. The linker will search the path specified by the --search_path option and then the C6X_C_DIR environment variable for the named library. You can use the --library linker option on the command line or in a command file.

cl6x --run_linker {**--rom_model** | **--ram_model**} *filenames* **--library=** *libraryname*

6.3.1.3 Library Order for Searching for Symbols

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.

6.3.2 Run-Time Initialization

C/C++ programs require initialization of the run-time environment before execution of the program itself may begin. This initialization is performed by a *bootstrap routine*. This routine is responsible for creating the stack, initializing global variables, and calling the main() function. The bootstrap routine should be the entry point for the program, and it typically should be the RESET interrupt handler. The bootstrap routine is responsible for the following tasks:

1. Set up the stack by initializing SP
2. Set up the data page pointer DP (for architectures that have one)
3. Set configuration registers
4. Process the *.cinit* table to autoinitialize global variables (when using the --rom_model option)
5. Process the *.pinit* table to construct global C++ objects.
6. Call the main() function with appropriate arguments
7. Call exit() when main() returns

When you compile a C/C++ program and use --rom_model or --ram_model, the linker automatically looks for a bootstrap routine named _c_int00. The run-time support library provides a sample _c_int00 in boot.c.obj, which performs the required tasks. If you use the run-time support's bootstrap routine, you should set _c_int00 as the entry point.

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. If your command line does not list any C/C++ files to compile and does not specify either the --ram_model or --rom_model link option, the linker does not know whether or not to use the C/C++ conventions, and you will receive a linker warning that says "warning: no suitable entry-point found; setting to 0". See Section 6.3.4 for details about using the --rom_model and --ram_model options.

6.3.3 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.

Constructors for global objects from a single module are invoked in the order declared in the source code, but the relative order of objects from different object files is unspecified.

Global constructors are called after initialization of other global variables and before the main() function is called. Global destructors are invoked during the exit run-time support function, similar to functions registered through atexit.

Section 8.9.2.6 discusses the format of the global constructor table.

6.3.4 Specifying the Type of Global Variable Initialization

The C/C++ compiler produces data tables for initializing global variables. Section 8.9.2.4 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 8.9.2.3).
- Global variables are initialized at *load time*. Use the --ram_model linker option (see Section 8.9.2.5).

If you use the linker command line without compiling any C/C++ files, you must use either the --rom_model or --ram_model option. These options tell the linker two things. First, they indicate that the linker should follow C/C++ conventions, using the definition of main() to link in the c_int00 boot routines. Second, they tell the linker

whether to select initialization at run time or load time. If your command line fails to include one of these options when it is required, you will see "warning: no suitable entry-point found; setting to 0".

If you use a single command line to both compile and link, the --rom_model option is the default. If used, the --rom_model or --ram_model option must follow the --run_linker option (see Section 6.1).

For details on linking conventions for EABI with --rom_model and --ram_model, see Section 8.9.2.3 and Section 8.9.2.5, respectively.

6.3.5 Specifying Where to Allocate Sections in Memory

The compiler produces relocatable blocks of code and data. These blocks, called *sections*, are allocated in memory in a variety of ways to conform to a variety of system configurations. See Section 8.1.1 for a complete description of how the compiler uses these sections.

The compiler creates two basic kinds of sections: initialized and uninitialized. Table 6-1 summarizes the initialized sections. Table 6-2 summarizes the uninitialized sections.

Table 6-1. Initialized Sections Created by the Compiler

Name	Contents
.args	Reserved space for copying command line arguments before the main() function is called by the boot routine. See Section 3.6.
.binit	Boot time copy tables (See the *TMS320C6000 Assembly Language Tools User's Guide* for information on BINIT in linker command files.)
.c6xabi.exidx	Index table for exception handling; read-only (see --exceptions option).
.c6xabi.extab	Unwinding instructions for exception handling; read-only (see --exceptions option).
.cinit	The compiler does not generate a .cinit section unless the --rom_mode linker option is specified. If --rom_mode is specified, the linker creates this section, which contains tables for explicitly initialized global and static variables.
.const	Global and static const variables, including string constants and initializers for local variables.
.data	Global and static non-const variables that are explicitly initialized.
.fardata	Far non-const global and static variables that are explicitly initialized.
.init_array	Table of constructors to be called at startup.
.name.load	Compressed image of section *name*; read-only (See the *TMS320C6000 Assembly Language Tools User's Guide* for information on copy tables.)
.neardata	Near non-const global and static variables that are explicitly initialized.
.ovly	Copy tables other than boot time (.binit) copy tables. Read-only data.
.ppdata	Data tables for compiler-based profiling (see the --gen_profile_info option).
.ppinfo	Correlation tables for compiler-based profiling (see the --gen_profile_info option).
.rodata	Global and static variables that have near and const qualifiers.
.switch	Jump tables for large switch statements.
.text	Standard default section for executable code. The --gen_func_subsections option causes code to be placed in separate .text:*func* section for each function, func().
.TI.crctab	Generated CRC checking tables. Read-only data.

Table 6-2. Uninitialized Sections Created by the Compiler

Name	Contents
.bss	Uninitialized global and static variables
.cio	Buffers for stdio functions from the run-time support library
.far	Global and static variables declared far
.stack	Stack
.sysmem	Memory pool (heap) for dynamic memory allocation (malloc, etc)

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 code

sections, the initialized and uninitialized sections created by the compiler cannot be allocated into internal program memory.

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*.

6.3.6 A Sample Linker Command File

Linker Command File shows a typical linker command file that links a C program. The command file in this example is named lnk.cmd and lists several linker options:

--rom_model	Tells the linker to use autoinitialization at run time.
--heap_size	Tells the linker to set the C heap size at 0x2000 bytes.
--stack_size	Tells the linker to set the stack size to 0x0100 bytes.
--library	Tells the linker to use an archive library file, rts64plus.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 *TMS320C6000 Assembly Language Tools User's Guide* for more information on these directives.

Linker Command File

```
--rom_model
--heap_size=0x2000
--stack_size=0x0100
--library=rts64plus.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
}
```

Chapter 7

C/C++ Language Implementation

The C language supported by the C6000 was developed by a committee of the American National Standards Institute (ANSI) and subsequently adopted by the International Standards Organization (ISO).

The C++ language supported by the C6000 is defined by the ANSI/ISO/IEC 14882:2014 standard with certain exceptions.

7.1 Characteristics of TMS320C6000 C

The C compiler supports the 1989, 1999, and 2011 versions of the C language:

- **C89.** Compiling with the --c89 option causes the compiler to conform to the ISO/IEC 9899:1990 C standard, which was previously ratified as ANSI X3.159-1989. The names "C89" and "C90" refer to the same programming language. "C89" is used in this document.
- **C99.** Compiling with the --c99 option causes the compiler to conform to the ISO/IEC 9899:1999 C standard.
- **C11.** Compiling with the --c11 option causes the compiler to conform to the ISO/IEC 9899:2011 C standard.

The C language is also described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R). The compiler can also accept many of the language extensions found in the GNU C compiler (see Section 7.14).

The compiler supports some features of C99 and C11 in the default relaxed ANSI mode with C89 support. It supports all language features of C99 in C99 mode and all language features of C11 in C11 mode. See Section 7.13.

The atomic operations described in the C11 standard are *not* supported.

The ANSI/ISO standard identifies some features of the C language that may be affected by characteristics of the target processor, run-time environment, or host environment. This set of features can differ among standard compilers.

Unsupported features of the C library are:

- The run-time library has minimal support for wide characters. The type wchar_t is implemented as unsigned short (16 bits), but can be an int if you set the --wchar_t=32 option. 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. See Section 7.4 for information about extended and multibyte character sets.
- 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.
- Some run-time functions and features in the C99/C11 specifications are not supported. See Section 7.13.

7.1.1 Implementation-Defined Behavior

The C standard requires that conforming implementations provide documentation on how the compiler handles instances of implementation-defined behavior.

The TI compiler officially supports a freestanding environment. The C standard does not require a freestanding environment to supply every C feature; in particular the library need not be complete. However, the TI compiler strives to provide most features of a hosted environment.

The section numbers in the lists that follow correspond to section numbers in Appendix J of the C99 standard. The numbers in parentheses at the end of each item are sections in the C99 standard that discuss the topic. Certain items listed in Appendix J of the C99 standard have been omitted from this list.

J.3.1 Translation

- The compiler and related tools emit diagnostic messages with several distinct formats. Diagnostic messages are emitted to stderr; any text on stderr may be assumed to be a diagnostic. If any errors are present, the tool will exit with an exit status indicating failure (non-zero). (3.10, 5.1.1.3)
- Nonempty sequences of white-space characters are preserved and are not replaced by a single space character in translation phase 3. (5.1.1.2)

J.3.2 Environment

- The compiler does not support multibyte characters in identifiers, string literals, or character constants. There is no mapping from multibyte characters to the source character set. However, the compiler accepts multibyte characters in comments. See Section 7.4 for details (5.1.1.2)
- The name of the function called at program startup is "main". (5.1.2.1)
- Program termination does not affect the environment; there is no way to return an exit code to the environment. By default, the program is known to have halted when execution reaches the special C$$EXIT label. (5.1.2.1)
- In relaxed ANSI mode, the compiler accepts "void main(void)" and "void main(int argc, char *argv[])" as alternate definitions of main. The alternate definitions are rejected in strict ANSI mode. (5.1.2.2.1)
- If space is provided for program arguments at link time with the --args option and the program is run under a system that can populate the .args section (such as CCS), argv[0] will contain the filename of the executable, argv[1] through argv[argc-1] will contain the command-line arguments to the program, and argv[argc] will be NULL. Otherwise, the value of argv and argc are undefined. (5.1.2.2.1)
- Interactive devices include stdin, stdout, and stderr (when attached to a system that honors CIO requests). Interactive devices are not limited to those output locations; the program may access hardware peripherals that interact with the external state. (5.1.2.3)
- Signals are not supported. The function signal is not supported. (7.14, 7.14.1.1)
- The library function getenv is implemented through the CIO interface. If the program is run under a system that supports CIO, the system performs getenv calls on the host system and passes the result back to the program. Otherwise the operation of getenv is undefined. No method of changing the environment from inside the target program is provided. (7.20.4.5)
- The system function is not supported. (7.20.4.6)

J.3.3. Identifiers

- The compiler does not support multibyte characters in identifiers. See Section 7.4 for details. (6.4.2)
- The number of significant initial characters in an identifier is unlimited. (5.2.4.1, 6.4.2)

J.3.4 Characters

- The number of bits in a byte (CHAR_BIT) is 8. See Section 7.3 for details about data types. (3.6)
- The execution character set is the same as the basic execution character set: plain ASCII. (5.2.1)
- The values produced for the standard alphabetic escape sequences are as follows. (5.2.2):

Escape Sequence	ASCII Meaning	Integer Value
\a	BEL (bell)	7
\b	BS (backspace)	8
\f	FF (form feed)	12
\n	LF (line feed)	10
\r	CR (carriage return)	13
\t	HT (horizontal tab)	9
\v	VT (vertical tab)	11

- The value of a char object into which any character other than a member of the basic execution character set has been stored is the ASCII value of that character. (6.2.5)
- Plain char is identical to signed char. (6.2.5, 6.3.1.1)
- The source character set and execution character set are both plain ASCII, so the mapping between them is one-to-one. The compiler accepts multibyte characters in comments. See Section 7.4 for details. (6.4.4.4, 5.1.1.2)
- The compiler currently supports only one locale, "C". (6.4.4.4)

- The compiler currently supports only one locale, "C". (6.4.5)

J.3.5 Integers

- C6000 supports the additional integer types __int40_t and unsigned __int40_t, which are signed and unsigned 40-bit integer types. (6.2.5)
- Negative values for signed integer types are represented as two's complement, and there are no trap representations. (6.2.6.2)
- The rank of __int40_t and unsigned __int40_t is less than the rank for long long. The rank of __int40_t and unsigned __int40_t is greater than the rank for long. (6.3.1.1)
- When an integer is converted to a signed integer type which cannot represent the value, the value is truncated (without raising a signal) by discarding the bits which cannot be stored in the destination type; the lowest bits are not modified. (6.3.1.3)
- Right shift of a signed integer value performs an arithmetic (signed) shift. The bitwise operations other than right shift operate on the bits in exactly the same way as on an unsigned value. That is, after the usual arithmetic conversions, the bitwise operation is performed without regard to the format of the integer type, in particular the sign bit. (6.5)

J.3.6 Floating point

- The accuracy of floating-point operations (+ - * /) is bit-exact. The accuracy of library functions that return floating-point results is not specified. (5.2.4.2.2)
- The compiler does not provide non-standard values for FLT_ROUNDS. (5.2.4.2.2)
- The compiler does not provide non-standard negative values of FLT_EVAL_METHOD. (5.2.4.2.2)
- The rounding direction when a floating-point number is converted to a narrower floating-point number is IEEE-754 "round to even". (6.3.1.5)
- For floating-point constants that are not exactly representable, the implementation uses the nearest representable value. (6.4.4.2)
- The compiler does not contract float expressions. (6.5)
- The default state for the FENV_ACCESS pragma is off. (7.6.1)
- The TI compiler does not define any additional float exceptions. (7.6, 7.12)
- The default state for the FP_CONTRACT pragma is off. (7.12.2)
- The "inexact" floating-point exception cannot be raised if the rounded result equals the mathematical result. (F.9)
- The "underflow" and "inexact" floating-point exceptions cannot be raised if the result is tiny but not inexact. (F.9)

J.3.7 Arrays and pointers

- When converting a pointer to an integer or vice versa, the pointer is considered an unsigned integer of the same size, and the normal integer conversion rules apply.
- When converting a pointer to an integer or vice versa, if the bitwise representation of the destination can hold all of the bits in the bitwise representation of the source, the bits are copied exactly. (6.3.2.3)
- The size of the result of subtracting two pointers to elements of the same array is the size of ptrdiff_t, which is defined in Section 7.3. (6.5.6)

J.3.8 Hints

- When the optimizer is used, the register storage-class specifier is ignored. When the optimizer is not used, the compiler will preferentially place register storage class objects into registers to the extent possible. The compiler reserves the right to place any register storage class object somewhere other than a register. (6.7.1)
- The inline function specifier is ignored unless the optimizer is used. For other restrictions on inlining, see Section 3.11.2. (6.7.4)

J.3.9 Structures, unions, enumerations, and bit-fields

- A "plain" int bit-field is treated as a signed int bit-field. (6.7.2, 6.7.2.1)
- In addition to _Bool, signed int, and unsigned int, the compiler allows char, signed char, unsigned char, signed short, unsigned shot, signed long, unsigned long, signed long long, unsigned long long, and enum types as bit-field types. (6.7.2.1)

- Bit-fields may not straddle a storage-unit boundary. (6.7.2.1)
- Bit-fields are allocated in endianness order within a unit. See Section 8.2.2. (6.7.2.1)
- Non-bit-field members of structures are aligned as specified in Section 8.2.1. (6.7.2.1)
- The integer type underlying each enumerated type is described in Section 7.3.1. (6.7.2.2)

J.3.10 Qualifiers

- The TI compiler does not shrink or grow volatile accesses. It is the user's responsibility to make sure the access size is appropriate for devices that only tolerate accesses of certain widths. The TI compiler does not change the number of accesses to a volatile variable unless absolutely necessary. This is significant for read-modify-write expressions such as += ; for an architecture which does not have a corresponding read-modify-write instruction, the compiler will be forced to use two accesses, one for the read and one for the write. Even for architectures with such instructions, it is not guaranteed that the compiler will be able to map such expressions to an instruction with a single memory operand. It is not guaranteed that the memory system will lock that memory location for the duration of the instruction. In a multi-core system, some other core may write the location after a RMW instruction reads it, but before it writes the result. The TI compiler will not reorder two volatile accesses, but it may reorder a volatile and a non-volatile access, so volatile cannot be used to create a critical section. Use some sort of lock if you need to create a critical section. (6.7.3)

J.3.11 Preprocessing directives

- Include directives may have one of two forms, " " or < >. For both forms, the compiler will look for a real file on-disk by that name using the include file search path. See Section 3.5.2. (6.4.7)
- The value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (both are ASCII). (6.10.1)
- The compiler uses the file search path to search for an included < > delimited header file. See Section 3.5.2. (6.10.2)
- The compiler uses the file search path to search for an included " " delimited header file. See Section 3.5.2. (6.10.2)
- There is no arbitrary nesting limit for #include processing. (6.10.2)
- See Section 7.9 for a description of the recognized non-standard pragmas. (6.10.6)
- The date and time of translation are always available from the host. (6.10.8)

J.3.12 Library functions

- Almost all of the library functions required for a hosted implementation are provided by the TI library, with exceptions noted in Section 7.13.1. (5.1.2.1)
- The format of the diagnostic printed by the assert macro is "Assertion failed, (*assertion macro argument*), file *file*, line *line*". (7.2.1.1)
- No strings other than "C" and "" may be passed as the second argument to the setlocale function. (7.11.1.1)
- No signal handling is supported. (7.14.1.1)
- The +INF, -INF, +inf, -inf, NAN, and nan styles can be used to print an infinity or NaN. (7.19.6.1, 7.24.2.1)
- The output for %p conversion in the fprintf or fwprintf function is the same as %x of the appropriate size. (7.19.6.1, 7.24.2.1)
- The termination status returned to the host environment by the abort, exit, or _Exit function is not returned to the host environment. (7.20.4.1, 7.20.4.3, 7.20.4.4)
- The system function is not supported. (7.20.4.6)

J.3.13 Architecture

- The values or expressions assigned to the macros specified in the headers float.h, limits.h, and stdint.h are described along with the sizes and format of integer types are described in Section 7.3. (5.2.4.2, 7.18.2, 7.18.3)
- The number, order, and encoding of bytes in any object are described in Section 8.2.1. (6.2.6.1)
- The value of the result of the sizeof operator is the storage size for each type, in terms of bytes. See Section 8.2.1. (6.5.3.4)

7.2 Characteristics of TMS320C6000 C++

The C6000 compiler supports C++ as defined in the ANSI/ISO/IEC 14882:2014 standard (C++14), including these features:

- Complete C++ standard library support, with exceptions noted below.
- Templates
- Exceptions, which are enabled with the --exceptions option; see Section 7.6.
- Run-time type information (RTTI), which can be enabled with the --rtti compiler option.

The compiler supports the 2014 standard of C++ as standardized by the ISO. However, the following features are *not* implemented or fully supported:

- The compiler does not support embedded C++ run-time-support libraries.
- The library supports wide chars (wchar_t), in that template functions and classes that are defined for char are also available for wchar_t. 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.
- No support for bad_cast or bad_type_id is included in the typeinfo header.
- Constant expressions for target-specific types are only partially supported.
- New character types (introduced in the C++11 standard) are not supported.
- Unicode string literals (introduced in the C++11 standard) are not supported.
- Universal character names in literals (introduced in the C++11 standard) are not supported.
- Atomic operations (introduced in the C++11 standard) are not supported.
- Data-dependency ordering for atomics and memory model (introduced in the C++11 standard) is not supported.
- Allowing atomics in signal handlers (introduced in the C++11 standard) is not supported.
- Strong compare and exchange (introduced in the C++11 standard) are not supported.
- Bidirectional fences (introduced in the C++11 standard) are not supported.
- Memory model (introduced in the C++11 standard) is not supported.
- Propagating exceptions (introduced in the C++11 standard) is not supported.
- Thread-local storage (introduced in the C++11 standard) is not supported.
- Dynamic initialization and destruction with concurrency (introduced in the C++11 standard) is not supported.

The changes made in order to support C++14 may cause "undefined symbol" errors to occur if you link with a C++ object file or library that was compiled with an older version of the compiler. If such linktime errors occur, recompile your C++ code using the --no_demangle command-line option. If any undefined symbol names begin with _Z or _ZVT, recompile the entire application, including object files and libraries. If you do not have source code for the libraries, download a newly-compiled version of the library.

7.3 Data Types

Table 7-1 lists the size, representation, and range of each scalar data type for the C6000 compiler. Many of the range values are available as standard macros in the header file limits.h.

The storage and alignment of data types is described in Section 8.2.1.

Table 7-1. TMS320C6000 C/C++ Data Types

			Range	
Type	**Size**	**Representation**	**Minimum**	**Maximum**
char, signed char	8 bits	ASCII	-128	127
unsigned char	8 bits	ASCII	0	255
_Bool, bool	8 bits	ASCII	0 (false)	1 (true)
short	16 bits	Binary	-32 768	32 767
unsigned short, wchar_t (1)	16 bits	Binary	0	65 535
int, signed int	32 bits	Binary	-2 147 483 648	2 147 483 647
unsigned int	32 bits	Binary	0	4 294 967 295
long, signed long	32 bits	Binary	-2 147 483 648	2 147 483 648
unsigned long	32 bits	Binary	0	4 294 967 295
__int40_t	40 bits	Binary	-549 755 813 888	549 755 813 887
unsigned __int40_t	40 bits	Binary	0	1 099 511 627 775
long long, signed long long	64 bits	Binary	-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 (2)	varies	Binary	varies	varies
float	32 bits	IEEE 32-bit	1.175 494e-38(3)	3.40 282 346e+38
float complex (4)	64 bits	Array of 2 IEEE 32-bit	1.175 494e-38 for real and imaginary portions separately	3.40 282 346e+38 for real and imaginary portions separately
double	64 bits	IEEE 64-bit	2.22 507 385e-308(3)	1.79 769 313e+308
double complex (4) (5)	128 bits	Array of 2 IEEE 64-bit	2.22 507 385e-308 for real and imaginary portions separately	1.79 769 313e+308 for real and imaginary portions separately
long double	64 bits	IEEE 64-bit	2.22 507 385e-308(3)	1.79 769 313e+308
long double complex (4) (5)	128 bits	Array of 2 IEEE 64-bit	2.22 507 385e-308 for real and imaginary portions separately	1.79 769 313e+308 for real and imaginary portions separately
pointers, references, pointer to data members	32 bits	Binary	0	0xFFFFFFFF

(1) This is the default type for wchar_t. You can use the --wchar_t option to change the wchar_t type to a 32-bit unsigned int type.
(2) For details about the size of an enum type, see Section 7.3.1.
(3) Figures are minimum precision.
(4) To use complex data types, you must include the <complex.h> header file. See Section 7.5.1 for more about complex data types.
(5) C6600 only

Negative values for signed types are represented using two's complement.

These additional types from C, C99 and C++ are defined as synonyms for standard types:

```
typedef unsigned int  size_t;
typedef int           ptrdiff_t;
typedef unsigned int  wchar_t;
typedef unsigned int  wint_t;
typedef char *        va_list;
```

7.3.1 Size of Enum Types

In the following declaration, `enum e` is an *enumerated type*. Each of `a` and `b` are *enumeration constants*.

```
enum e { a, b=N };
```

Each enumerated type is assigned an integer type that can hold all of the enumeration constants. This integer type is the "underlying type." The type of each enumeration constant is also an integer type, and in C might not be the same type. Be careful to note the difference between the *underlying type of an enumerated type* and the *type of an enumeration constant*.

The size and signedness chosen for the enumerated type and each enumeration constant depend on the values of the enumeration constants and whether you are compiling for C or C++. C++11 allows you to specify a specific type for an enumeration type; if such a type is provided, it will be used and the rest of this section does not apply.

In C++ mode, the compiler allows enumeration constants up to the largest integral type (64 bits). The C standard says that all enumeration constants in strictly conforming C code (C89/C99/C11) must have a value that fits into the type "int;" however, as an extension, you may use enumeration constants larger than "int" even in C mode.

For the enumerated type, the compiler selects the first type in this list that is big enough and of the correct sign to represent all of the values of the enumeration constants:

- unsigned char
- signed char
- unsigned short
- signed short
- unsigned int
- signed int
- unsigned long long
- signed long long

The "long" type is skipped because it is the same size as "int."

For example, this enumerated type will have "unsigned char" as its underlying type:

```
enum uc { a, b, c };
```

But this one will have "signed char" as its underlying type:

```
enum sc { a, b, c, d = -1 };
```

And this one will have "signed short" as its underlying type:

```
enum ss { a, b, c, d = -1, e = UCHAR_MAX };
```

For C++, the enumeration constants are all of the same type as the enumerated type.

For C, the enumeration constants are assigned types depending on their value. All enumeration constants with values that can fit into "int" are given type "int," even if the underlying type of the enumerated type is smaller than "int." All enumeration constants that do not fit in an "int" are given the same type as the underlying type of the enumerated type. This means that some enumeration constants may have a different size and signedness than the enumeration type.

7.3.2 Vector Data Types

The C/C++ compiler supports the use of TI vector data types in C/C++ source files. Vector data types are useful because they can make use of the natural vector width within the processing cores. Vector data types provide a straightforward way to utilize the SIMD instructions that are available on that architecture. Vector data types also provide a more direct mapping from the abstract model of a vector data object to the physical representation of that data object in a register.

Vector data types are similar to an array, in that a vector contains a specified number of elements of a specified type. However, the vector length can only be one of 2, 3, 4, 8, or 16. Wherever possible, intrinsics that act upon vectors are optimized to make use of efficient single instruction, multiple data (SIMD) instructions on the device.

You can enable support for vector data types by using the --vectypes compiler option.

All of the vector data types and related built-in functions that are supported in the C6000 programming model are specified in the "c6x_vec.h" header file in the "include" sub-directory where your C6000 CGT was installed. Any C/C++ source file that uses vector data types or any of the related built-in functions must contain the following in that source file:

```
#include <c6x_vec.h>
```

A vector type name concatenates an element type name and a number representing the vector length. The resulting vector consists of the specified number of elements of the specified type.

The C6000 implementation of vector data types and operations follows the OpenCL C language specification closely. For a detailed description of OpenCL vector data types and operations, please see version 1.2 of The OpenCL Specification, which is available from the Khronos OpenCL Working Group. Section 6.1.2 of the version 1.2 specification provides a detailed description of the built-in vector data types supported in the OpenCL C programming language. The C6000 programming model provides the following built-in vector data types:

Table 7-2. Vector Data Types

Type	Description	Maximum Elements
char*n*	A vector of *n* 8-bit signed integer values.	16
uchar*n*	A vector of *n* 8-bit unsigned integer values.	16
short*n*	A vector of *n* 16-bit signed integer values.	8
ushort*n*	A vector of *n* 16-bit unsigned integer values.	8
int*n*	A vector of *n* 32-bit signed integer values.	4
uint*n*	A vector of *n* 32-bit unsigned integer values.	4
longlong*n*	A vector of *n* 64-bit signed integer values.	2
ulonglong*n*	A vector of *n* 64-bit unsigned integer values.	2
float*n*	A vector of *n* 32-bit single-precision floating-point values.	4
double*n*	A vector of *n* 64-bit double-precision floating-point values.	8

where n can be a vector length of 2, 3, 4, 8, or 16.

For example, a "uchar8" is a vector of 8 unsigned chars; its length is 8 and its size is 64 bits. A "float4" is a vector of 4 float elements; its length is 4 and its size is 128 bits.

Vectors types are aligned on a boundary equal to the total size of the vector's elements up to 64 bits. Any vector type with a total size of more than 64 bits is aligned to a 64-bit boundary (8 bytes). For example, a short2 has a total size of 32 bits and is aligned on a 4-byte boundary. A longlong2 has a total size of 128 bits and is aligned on an 8-byte boundary.

Note

To avoid confusion between C6000's definition of long (32-bits) and 64-bit definitions of long, vector types with a base type of "long" and unsigned long ("ulong") are not provided. If you want to use the standard long and ulong types, you can create a simple preprocessor macro such as: `#define long2 longlong2` or `#define long2 int2` , depending the element type and size you want to use.

The Code Generation Tools also provide an extension for representing vectors of complex types. A prefix of 'c' is used to indicate a complex type name. Each complex type vector element contains a real part and an imaginary part with the real part occupying the lower address in memory. Thus, the complex vector types are as follows:

Table 7-3. Complex Vector Data Types

Type	Description	Maximum Elements
ccharn	A vector of *n* pairs of 8-bit signed integer values.	8
cshortn	A vector of *n* pairs of 16-bit signed integer values.	4
cintn	A vector of *n* pairs of 32-bit signed integer values.	2
clonglongn	A vector of *n* pairs of 64-bit signed integer values.	1
cfloatn	A vector of *n* pairs of 32-bit floating-point values.	2
cdoublen	A vector of *n* pairs of 64-bit floating-point values.	1

where n can be a vector length of 1, 2, 4, or 8. Note that 16 is not a valid vector length for complex vector types. For example, a "cfloat2" is a vector of 2 complex floats. Its length is 2 and its size is 128 bits. Each "cfloat2" vector element contains a real float and an imaginary float.

For information about operators and built-in functions used with vector data types, see Section 7.15.

7.4 File Encodings and Character Sets

The compiler accepts source files with one of two distinct encodings:

- **UTF-8 with Byte Order Mark (BOM).** These files may contain extended (multibyte) characters in C/C++ comments. In all other contexts—including string constants, identifiers, assembly files, and linker command files—only 7-bit ASCII characters are supported.
- **Plain ASCII files.** These files must contain only 7-bit ASCII characters.

To choose the UTF-8 encoding in Code Composer Studio, open the Preferences dialog, select **General > Workspace**, and set the **Text File Encoding** to UTF-8.

If you use an editor that does not have a "plain ASCII" encoding mode, you can use Windows-1252 (also called CP-1252) or ISO-8859-1 (also called Latin 1), both of which accept all 7-bit ASCII characters. However, the compiler may not accept extended characters in these encodings, so you should not use extended characters, even in comments.

Wide character (wchar_t) types and operations are supported by the compiler. However, wide character strings may not contain characters beyond 7-bit ASCII. The encoding of wide characters is 7-bit ASCII, 0 extended to the width of the wchar_t type.

7.5 Keywords

The C6000 C/C++ compiler supports all of the standard C89 keywords, including const, volatile, and register. It supports all of the standard C99 keywords, including inline and restrict. It supports all of the standard C11 keywords. It also supports TI extension keywords __interrupt,__near, __far,__cregister,and __asm. Some keywords are not available in strict ANSI mode.

The following keywords may appear in other target documentation and require the same treatment as the interrupt and restrict keywords:

- trap
- reentrant
- cregister

7.5.1 The complex Keyword

To use complex data types, you must include the <complex.h> header file. If this header file is included, complex support is available for all C/C++ modes, including relaxed and strict ANSI modes and C89 and C99. The <complex.h> header file implements math operation and function support for complex data types.

Complex types are implemented as an array of two elements. For example, for the following declaration, the variable is stored as an array of two floats. The real portion of the number is stored in x._Vals[0] and the imaginary portion of the number is stored in x._Vals[1].

```
float complex x;
```

7.5.2 The const Keyword

The C/C++ compiler supports the ANSI/ISO standard keyword *const* in all modes. This keyword gives you greater optimization and control over allocation 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.

Global objects qualified as far const are placed in the .const section. The linker allocates the .const section from ROM or FLASH, which are typically more plentiful than RAM. The const data storage allocation rule has the following exceptions:

- If *volatile* is also specified in the object definition. For example, `volatile const int x`. Volatile keywords are assumed to be allocated to RAM. (The program is not allowed to modify a const volatile object, but something external to the program might.)
- If the object has automatic storage (allocated on the stack).
- If the object is a C++ object with a "mutable" member.
- If the object is initialized with a value that is not known at compile time (such as the value of another variable).

In these cases, the storage for the object is the same as if the const keyword were not used.

The placement of the const keyword is important. For example, the first statement below defines a constant pointer p to a modifiable int. The second statement defines a modifiable pointer q to a constant int:

```
int * const p = &x;
const int * q = &x;
```

Using the const keyword, you can define large constant tables and allocate them into system ROM. For example, to allocate a ROM table, you could use the following definition:

```
far const int digits[] = {0,1,2,3,4,5,6,7,8,9};
```

7.5.3 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 (see Table 7-4). If the name matches, the compiler generates the code to reference the control register. If the name does not match, the compiler issues an error.

Table 7-4. Control Registers for C64x+, C6740, and C6600

Register	Description
AMR	Addressing mode register
CSR	Control status register
DNUM	DSP core number register
ECR	Exception clear register
EFR	Exception flag register
GFPGFR	Galois field multiply control register
GPLYA	GMPY A-side polynomial register
GPLYB	GMPY B-side polynomial register
ICR	Interrupt clear register
IER	Interrupt enable register
IERR	Internal exception report register
IFR	Interrupt flag register. (IFR is read only.)
ILC	Inner loop count register
IRP	Interrupt return pointer register
ISR	Interrupt set register
ISTP	Interrupt service table pointer register
ITSR	Interrupt task state register
NRP	Nonmaskable interrupt return pointer register
NTSR	NMI/exception task state register
PCE1	Program counter, E1 phase
REP	Restricted entry point address register
RILC	Reload inner loop count register
SSR	Saturation status register
TSCH	Time-stamp counter (high 32) register
TSCL	Time-stamp counter (low 32) register
TSR	Task state register

The additional control registers listed in Table 7-5 are used for floating-point operations on C6740 and C6600 devices:

Table 7-5. Additional Control Registers for C6740 and C6600

Register	Description
FADCR	Floating-point adder configuration register
FAUCR	Floating-point auxiliary configuration register
FMCR	Floating-point multiplier configuration 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 7-4, 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. See the *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide* (SPRU732), the *TMS320C66x DSP CPU and Instruction Set Reference Guide* (SPRUGH7), or the *TMS320C674x DSP CPU and Instruction Set Reference Guide* (SPRUFE8) for detailed information on the control registers.

See Example 7-1 for an example that declares and uses control registers.

Example 7-1. Define and Use Control Registers

```
extern __cregister volatile unsigned int AMR;
extern __cregister volatile unsigned int CSR;
extern __cregister volatile unsigned int IFR;
extern __cregister volatile unsigned int ISR;
extern __cregister volatile unsigned int ICR;
extern __cregister volatile unsigned int IER;
extern __cregister volatile unsigned int FADCR;
extern __cregister volatile unsigned int FAUCR;
extern __cregister volatile unsigned int FMCR;
main()
{
   printf("AMR = %x\n", AMR);
}
```

7.5.4 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. This keyword is an IRQ interrupt. The alternate keyword, "interrupt", may also be used except in strict ANSI C or C++ modes.

Note that the interrupt function attribute described in Section 7.9.20 is the recommended syntax for declaring interrupt functions.

Functions that handle interrupts follow special register-saving rules and a special return sequence. The implementation stresses safety. The interrupt routine does not assume that the C run-time conventions for the various CPU register and status bits are in effect; instead, it re-establishes any values assumed by the run-time environment. 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 main() function. Because it has no caller, c_int00 does not save any registers.

Note

Hwi Objects and the __interrupt Keyword: The __interrupt keyword must not be used when SYS/BIOS Hwi objects are used in conjunction with C functions. The Hwi_enter/Hwi_exit macros and the Hwi dispatcher already contain this functionality; use of the C modifier can cause unwanted conflicts.

7.5.5 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 specifiers. One can appear before, after, or in between the storage class specifiers and types. Normally only one storage class specifier is permitted in a variable declaration. However, two storage class specifiers can be used in a single declaration if one of the two is __near or __far.

The following examples are legal combinations of __near and __far with other storage class specifiers:

```
__far static int x;
static __near int x;
static int __far x;
__far int foo();
static __far int foo();
```

7.5.5.1 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
```

Be consistent with near and far declarations. If an object is defined to be far, all external declarations of this object in other C files or headers must also contain the __far keyword, or you will likely get compiler or linker errors. If an object is defined to be near, you can safely declare it as __far in other C files or headers, but you will have slower data access for that variable.

If you use the DATA_SECTION pragma, the object is indicated as a far variable, and this cannot be overridden. If you reference this object in another file, then you need to use *extern __far* when declaring this object in the other source file. This ensures access to the variable, since the variable might not be in the .bss section. For details, see Section 7.9.6.

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 8.1.4.1.

7.5.5.2 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 8.1.5.

7.5.6 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.

The "restrict" keyword is a C99 keyword, and cannot be accepted in strict ANSI C89 mode. Use the "__restrict" keyword if the strict ANSI C89 mode must be used. See Section 7.13.

In the following example, 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.

```
void func1(int * restrict a, int * restrict b)
{
  /* func1's code here */
}
```

The following example uses the restrict keyword when passing arrays to a function. Here, the arrays c and d must not overlap, nor may c and d point to the same array.

```
void func2(int c[restrict], int d[restrict])
{
  int i;
  for(i = 0; i < 64; i++)
  {
    c[i] += d[i];
    d[i] += 1;
  }
}
```

7.5.7 The volatile Keyword

The C/C++ compiler supports the *volatile* keyword in all modes. In addition, the __volatile keyword is supported in relaxed ANSI mode for C89, C99, C11, and C++.

The volatile keyword indicates to the compiler that there is something about how the variable is accessed that requires that the compiler not use overly-clever optimization on expressions involving that variable. For example, the variable may also be accessed by an external program, an interrupt, another thread, or a peripheral device.

The compiler eliminates redundant memory accesses whenever possible, using data flow analysis to figure out when it is legal. However, some memory accesses may be special in some way that the compiler cannot see, and in such cases you should use the volatile keyword to prevent the compiler from optimizing away something

important. The compiler does not optimize out any accesses to variables declared volatile. The number of volatile reads and writes will be exactly as they appear in the C/C++ code, no more and no less and in the same order.

Any variable which might be modified by something external to the obvious control flow of the program (such as an interrupt service routine) must be declared volatile. This tells the compiler that an interrupt function might modify the value at any time, so the compiler should not perform optimizations which will change the number or order of accesses of that variable. This is the primary purpose of the volatile keyword. In the following example, the loop intends to wait for a location to be read as 0xFF:

```
   unsigned int *ctrl;
   while (*ctrl !=0xFF);
```

However, in this example, *ctrl is a loop-invariant expression, so the loop is optimized down to a single-memory read. To get the desired result, define ctrl as:

```
   volatile unsigned int *ctrl;
```

Here the *ctrl pointer is intended to reference a hardware location, such as an interrupt flag.

The volatile keyword must also be used when accessing memory locations that represent memory-mapped peripheral devices. Such memory locations might change value in ways that the compiler cannot predict. These locations might change if accessed, or when some other memory location is accessed, or when some signal occurs.

Volatile must also be used for local variables in a function which calls setjmp, if the value of the local variables needs to remain valid if a longjmp occurs.

```
#include <stdlib.h>
jmp_buf context;
void function()
{
    volatile int x = 3;
    switch(setjmp(context))
    {
        case 0: setup(); break;
        default:
        {
           /* We only reach here if longjmp occurs. Because x's lifetime begins before setjmp
              and lasts through longjmp, the C standard requires x be declared "volatile". */
           printf("x == %d\n", x);
           break;
        }
    }
}
```

The --interrupt_threshold=1 option should be used when compiling with volatiles.

7.6 C++ Exception Handling

The compiler supports the C++ exception handling features defined by the ANSI/ISO 14882 C++ Standard. See *The C++ Programming Language, Third Edition* by Bjarne Stroustrup. The compiler's --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 that file. Mixing exception-enabled and exception-disabled object files and libraries can lead to undefined behavior.

Exception handling requires support in the run-time-support library, which come in exception-enabled and exception-disabled forms; you must link with the correct form. When using automatic library selection (the default), the linker automatically selects the correct library Section 6.3.1.1. If you select the library manually, you must use run-time-support libraries whose name contains _eh if you enable exceptions.

Using the --exceptions option causes the compiler to insert exception handling code. This code will increase the size of the program somewhat. In addition, there is a minimal execution time cost even if exceptions are never thrown, and a slight increase in the data size for the exception-handling tables.

See Section 9.1 for details on the run-time libraries.

7.7 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 8.3.

7.8 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 implemented through the ***__asm*** keyword. The __asm keyword provides access to hardware features that C/C++ cannot provide.

The alternate keyword, "asm", may also be used except in strict ANSI C mode. It is available in relaxed C and C++ modes.

Using __asm is syntactically performed as 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 *naked* function attribute can be used to identify functions that are written as embedded assembly functions using `__asm` statements. See Section 7.14.2.

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.

The __asm statement does not provide any way to refer to local variables. If your assembly code needs to refer to local variables, you will need to write the entire function in assembly code.

For more information, refer to Section 8.6.5.

Note

Avoid Disrupting the C/C++ Environment With asm Statements

Be careful not to disrupt the C/C++ environment with __asm statements. The compiler does not check the inserted instructions. Inserting jumps and labels into C/C++ code can cause unpredictable results in variables manipulated in or around the inserted code. Directives that change sections or otherwise affect the assembly environment can also be troublesome.

Be especially careful when you use optimization with __asm statements. Although the compiler cannot remove __asm statements, it can significantly rearrange the code order near them and cause undesired results.

7.9 Pragma Directives

The following pragma directives tell the compiler how to treat a certain function, object, or section of code.

- CALLS (See Section 7.9.1)
- CODE_ALIGN (See Section 7.9.2)
- CODE_SECTION (See Section 7.9.3)
- DATA_ALIGN (See Section 7.9.4)
- DATA_MEM_BANK (See Section 7.9.5)
- DATA_SECTION (See Section 7.9.6)
- diag_suppress, diag_remark, diag_warning, diag_error, diag_default, diag_push, diag_pop (See Section 7.9.7)
- FORCEINLINE (See Section 7.9.8)
- FORCEINLINE_RECURSIVE (See Section 7.9.9)
- FUNC_ALWAYS_INLINE (See Section 7.9.10)
- FUNC_CANNOT_INLINE (See Section 7.9.11)
- FUNC_EXT_CALLED (See Section 7.9.12)
- FUNC_INTERRUPT_THRESHOLD (See Section 7.9.13)
- FUNC_IS_PURE (See Section 7.9.14)
- FUNC_IS_SYSTEM (See Section 7.9.15)
- FUNC_NEVER_RETURNS (See Section 7.9.16)
- FUNC_NO_GLOBAL_ASG (See Section 7.9.17)
- FUNC_NO_IND_ASG (See Section 7.9.18)
- FUNCTION_OPTIONS (See Section 7.9.19)
- INTERRUPT (See Section 7.9.20)
- LOCATION (See Section 7.9.21)
- MUST_ITERATE (See Section 7.9.22)
- NMI_INTERRUPT (See Section 7.9.23)
- NOINIT (See Section 7.9.24)
- NOINLINE (See Section 7.9.25)
- NO_HOOKS (See Section 7.9.26)
- once (See Section 7.9.27)
- pack (See Section 7.9.28)
- PERSISTENT (See Section 7.9.24)
- PROB_ITERATE (See Section 7.9.29)
- RETAIN (See Section 7.9.30)
- SET_CODE_SECTION (See Section 7.9.31)
- SET_DATA_SECTION (See Section 7.9.31)
- STRUCT_ALIGN (See Section 7.9.32)
- UNROLL (See Section 7.9.33)

Additional pragmas are provided to support the OpenMP API (see Section 8.10.1). For descriptions of these pragmas, see the TI OpenMP Accelerator Model online documentation.

Most 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. You must specify the pragma outside the body of a function, and pragma specifications 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 and may ignore the pragma.

For pragmas that apply to functions or symbols, the syntax 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. Because the entity operated on is specified, a pragma in C can appear some distance way from the definition of that entity.
- In C++, pragmas are positional. They do not name the entity on which they operate as an argument. Instead, they always operate on the next entity defined after the pragma.

7.9.1 The CALLS Pragma

The CALLS pragma specifies a set of functions that can be called indirectly from a specified calling function.

The CALLS pragma is used by the compiler to embed debug information about indirect calls in object files. Using the CALLS pragma on functions that make indirect calls enables such indirect calls to be included in calculations for such functions' inclusive stack sizes. For more information on generating function stack usage information, see the -cg option of the Object File Display Utility in the "Invoking the Object File Display Utility" section of the *TMS320C6000 Assembly Language Tools User's Guide*.

The CALLS pragma can precede either the calling function's definition or its declaration. In C, the pragma must have at least 2 arguments—the first argument is the calling function, followed by at least one function that will be indirectly called from the calling function. In C++, the pragma applies to the next function declared or defined, and the pragma must have at least one argument.

The syntax for the CALLS pragma in C is as follows. This indicates that calling_function can indirectly call function_1 through function_n.

#pragma CALLS (*calling_function*, *function_1*, *function_2*, ..., *function_n* **)**

The syntax for the CALLS pragma in C++ is:

#pragma CALLS (*function_1_mangled_name*, ..., *function_n_mangled_name* **)**

Note that in C++, the arguments to the CALLS pragma must be the full mangled names for the functions that can be indirectly called from the calling function.

The GCC-style "calls" function attribute (see Section 7.14.2), which has the same effect as the CALLS pragma, has the following syntax:

```
__attribute__((calls("function_1","function_2",..., "function_n")))
```

7.9.2 The CODE_ALIGN Pragma

The CODE_ALIGN pragma aligns *func* along the specified alignment. The alignment *constant* must be a power of 2. The CODE_ALIGN pragma is useful if you have functions that you want to start at a certain boundary.

The CODE_ALIGN pragma has the same effect as using the GCC-style `aligned` function attribute. See Section 7.14.2.

The syntax of the pragma in C is:

#pragma CODE_ALIGN (*func* **,** *constant* **)**

The syntax of the pragma in C++ is:

#pragma CODE_ALIGN (*constant* **)**

7.9.3 The CODE_SECTION Pragma

The CODE_SECTION pragma allocates space for the *symbol* in C, or the next symbol declared in C++, in a section named *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 CODE_SECTION pragma has the same effect as using the GCC-style `section` function attribute. See Section 7.14.2.

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 following example demonstrates the use of the CODE_SECTION pragma.

Using the CODE_SECTION Pragma in C

```
#pragma CODE_SECTION(fn, "my_sect")
int fn(int x)
{
    return x;
}
```

This example C code results in the following generated assembly code:

```
    .sect    "my_sect"
    .global  _fn
;******************************************************************************
;* FUNCTION NAME: _fn                                                         *
;*                                                                            *
;*   Regs Modified     : SP                                                   *
;*   Regs Used         : A4,B3,SP                                             *
;*   Local Frame Size  : 0 Args + 4 Auto + 0 Save = 4 byte                    *
;******************************************************************************
_fn:
;** --------------------------------------------------------------------------*
           RET     .S2     B3                ; |6|
           SUB     .D2     SP,8,SP           ; |4|
           STW     .D2T1   A4,*+SP(4)        ; |4|
           ADD     .S2     8,SP,SP           ; |6|
           NOP             2
           ; BRANCH OCCURS                   ; |6|
```

7.9.4 The DATA_ALIGN Pragma

The DATA_ALIGN pragma aligns the *symbol* in C, or the next symbol declared in C++, 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 maximum alignment is 32768.

The DATA_ALIGN pragma cannot be used to reduce an object's natural alignment.

Using the DATA_ALIGN pragma has the same effect as using the GCC-style `aligned` variable attribute. See Section 7.14.4.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma DATA_ALIGN (*constant* **)**

7.9.5 The DATA_MEM_BANK Pragma

The DATA_MEM_BANK pragma aligns a symbol or variable to a specified internal data memory bank boundary. The *constant* specifies a specific memory bank to start your variables on. (See Section 5.5 for a graphic representation of memory banks.) C6400+, C6740, and C6600 devices contain 8 memory banks. The value of *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* **)**

Only global variables can be aligned with the DATA_MEM_BANK pragma.

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.

A value of 0 for the constant argument to DATA_MEM_BANK pragma causes the last five bits of the starting address to be 0x00. For a value of 2, the last five bits of the starting address will be 0x08 (0b01000). For a value of 4, the last five bits of the starting address will be 0x10 (0b10000). For a value of 6, the last five bits of the starting address will be 0x18 (0b11000).

The code in Example 7-2 uses the DATA_MEM_BANK pragma to specify the alignment of the x, y, z, w, and zz arrays. It then assigns values to all the array elements and prints the starting address of each array.

Example 7-2. Using the DATA_MEM_BANK Pragma

```
#include <stdio.h>
#pragma DATA_MEM_BANK (x, 2)
short x[100];
#pragma DATA_MEM_BANK (z, 0)
#pragma DATA_SECTION (z, ".z_sect")
short z[100];
#pragma DATA_MEM_BANK (w, 4)
#pragma DATA_SECTION (w, ".w_sect")
short w[100];
#pragma DATA_MEM_BANK (zz, 6)
#pragma DATA_SECTION (zz, ".zz_sect")
short zz[100];
static short my_count = 0;
void main()
{
   int i;
   #pragma DATA_MEM_BANK (y, 4)
   short y[100];
   for (i = 0; i < 100; i++)
   {
      w[i] = my_count++;
      x[i] = my_count++;
      y[i] = my_count++;
      z[i] = my_count++;
      zz[i] = my_count++;
   }
   printf("address of  w: 0x%08lx\n", (unsigned long)w);
   printf("address of  x: 0x%08lx\n", (unsigned long)x);
   printf("address of  y: 0x%08lx\n", (unsigned long)y);
   printf("address of  z: 0x%08lx\n", (unsigned long)z);
   printf("address of zz: 0x%08lx\n", (unsigned long)zz);
}
```

Sample output is as follows:

```
address of  w: 0x00006a70
address of  x: 0x80009468
address of  y: 0x80005f10
address of  z: 0x00006b60
address of zz: 0x00006978
```

7.9.6 The DATA_SECTION Pragma

The DATA_SECTION pragma allocates space for the *symbol* in C, or the next symbol declared in C++, in a section named *section name*. This 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.

Using the DATA_SECTION pragma has the same effect as using the GCC-style `section` variable attribute. See Section 7.14.4.

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* **")**

Example 7-3 through Example 7-5 demonstrate the use of the DATA_SECTION pragma.

Example 7-3. Using the DATA_SECTION Pragma C Source File

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

Example 7-4. Using the DATA_SECTION Pragma C++ Source File

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

Example 7-5. Using the DATA_SECTION Pragma Assembly Source File

```
            .global _bufferA
            .bss    _bufferA,512,4
            .global _bufferB
```

7.9.7 The Diagnostic Message Pragmas

The following pragmas can be used to control diagnostic messages in the same ways as the corresponding command line options:

Pragma	Option	Description
diag_suppress *num*	-pds=*num*[, num_2, num_3...]	Suppress diagnostic *num*
diag_remark *num*	-pdsr=*num*[, num_2, num_3...]	Treat diagnostic *num* as a remark
diag_warning *num*	-pdsw=*num*[, num_2, num_3...]	Treat diagnostic *num* as a warning
diag_error *num*	-pdse=*num*[, num_2, num_3...]	Treat diagnostic *num* as an error
diag_default *num*	n/a	Use default severity of the diagnostic
diag_push	n/a	Push the current diagnostics severity state to store it for later use.
diag_pop	n/a	Pop the most recent diagnostic severity state stored with #pragma diag_push to be the current setting.

The syntax of the diag_suppress, diag_remark, diag_warning, and diag_error pragmas in C is:

#pragma diag_ *xxx* [=]*num*[, num_2, num_3...]

Notice that the names of these pragmas are in lowercase.

The diagnostic affected (*num*) is specified using either an error number or an error tag name. The equal sign (=) is optional. Any diagnostic can be overridden to be an error, but only diagnostic messages with a severity of discretionary error or below can have their severity reduced to a warning or below, or be suppressed. The diag_default pragma is used to return the severity of a diagnostic to the one that was in effect before any pragmas were issued (i.e., the normal severity of the message as modified by any command-line options).

The diagnostic identifier number is output with the message when you use the -pden command line option.

7.9.8 The FORCEINLINE Pragma

The FORCEINLINE pragma can be placed before a statement to force any function calls made in that statement to be inlined. It has no effect on other calls to the same functions.

The compiler only inlines a function if it is legal to inline the function. Functions are never inlined if the compiler is invoked with the --opt_level=off option. A function can be inlined even if the function is not declared with the inline keyword. A function can be inlined even if the compiler is not invoked with any --opt_level command-line option.

The syntax of the pragma in C/C++ is:

```
#pragma FORCEINLINE
```

For example, in the following example, the mytest() and getname() functions are inlined, but the error() function is not.

```
#pragma FORCEINLINE
if (!mytest(getname(myvar))) {
    error();
}
```

Placing the FORCEINLINE pragma before the call to error() would inline that function but not the others.

For information about interactions between command-line options, pragmas, and keywords that affect inlining, see Section 3.11.

Notice that the FORCEINLINE, FORCEINLINE_RECURSIVE, and NOINLINE pragmas affect only the C/C++ statement that follows the pragma. The FUNC_ALWAYS_INLINE and FUNC_CANNOT_INLINE pragmas affect an entire function.

7.9.9 The FORCEINLINE_RECURSIVE Pragma

The FORCEINLINE_RECURSIVE can be placed before a statement to force any function calls made in that statement to be inlined along with any calls made from those functions. That is, calls that are not visible in the statement but are called as a result of the statement will be inlined.

The syntax of the pragma in C/C++ is:

```
#pragma FORCEINLINE_RECURSIVE
```

For information about interactions between command-line options, pragmas, and keywords that affect inlining, see Section 3.11.

7.9.10 The FUNC_ALWAYS_INLINE Pragma

The FUNC_ALWAYS_INLINE pragma instructs the compiler to always inline the named function.

The compiler only inlines a function if it is legal to inline the function. Functions are never inlined if the compiler is invoked with the --opt_level=off option. A function can be inlined even if the function is not declared with the inline keyword. A function can be inlined even if the compiler is not invoked with any --opt_level command-line option. See Section 3.11 for details about interaction between various types of inlining.

This 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 FUNC_ALWAYS_INLINE pragma has the same effect as using the GCC-style `always_inline` function attribute. See Section 7.14.2.

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

The following example uses this pragma:

```
#pragma FUNC_ALWAYS_INLINE(functionThatMustGetInlined)
static inline void functionThatMustGetInlined(void) {
    P1OUT |= 0x01;
    P1OUT &= ~0x01;
}
```

Note

Use Caution with the FUNC_ALWAYS_INLINE Pragma

The FUNC_ALWAYS_INLINE pragma overrides the compiler's inlining decisions. Overuse of this pragma could result in increased compilation times or memory usage, potentially enough to consume all available memory and result in compilation tool failures.

7.9.11 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 3.11.

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 FUNC_CANNOT_INLINE pragma has the same effect as using the GCC-style `noinline` function attribute. See Section 7.14.2.

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
```

7.9.12 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 that the optimizer should keep these C functions or any functions 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 to keep. In C, the argument *func* is the name of the function to keep. 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 4.4.2.

7.9.13 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 3.12). 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)
```

7.9.14 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

7.9.15 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
```

7.9.16 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
```

7.9.17 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
```

7.9.18 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
```

7.9.19 The FUNCTION_OPTIONS Pragma

The FUNCTION_OPTIONS pragma allows you to compile a specific function in a C or C++ file with additional command-line compiler options. The affected function will be compiled as if the specified list of options appeared on the command line after all other compiler options. In C, the pragma is applied to the function specified. In C++, the pragma is applied to the next function.

The syntax of the pragma in C is:

```
#pragma FUNCTION_OPTIONS ( func , " additional options " )
```

The syntax of the pragma in C++ is:

```
#pragma FUNCTION_OPTIONS( " additional options " )
```

Supported options for this pragma are --opt_level, --auto_inline, --code_state, --opt_for_space, and --opt_for_speed.

In order to use --opt_level and --auto_inline with the FUNCTION_OPTIONS pragma, the compiler must be invoked with some optimization level (that is, at least --opt_level=0). The FUNCTION_OPTIONS pragma is ignored if --opt_level=off. The FUNCTION_OPTIONS pragma cannot be used to completely disable the optimizer for the compilation of a function; the lowest optimization level that can be specified is --opt_level=0.

7.9.20 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
void func ( void )
```

```
__attribute__((interrupt )) void func ( void )
```

The code for the function will return via the IRP (interrupt return pointer).

```
#pragma INTERRUPT ( func , {HPI|LPI} )
```

```
#pragma INTERRUPT ( {HPI|LPI} )
```

Note

Hwi Objects and the INTERRUPT Pragma: The INTERRUPT pragma must not be used when SYS/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 negative results.

7.9.21 The LOCATION Pragma

The compiler supports the ability to specify the run-time address of a variable at the source level. This can be accomplished with the LOCATION pragma or the GCC-style location attribute. The LOCATION pragma has the same effect as using the GCC-style `location` function attribute. See Section 7.14.2.

The syntax of the pragma in C is:

```
#pragma LOCATION( x , address )
int x
```

The syntax of the pragmas in C++ is:

```
#pragma LOCATION( address )
int x
```

The syntax of the GCC-style attribute (see Section 7.14.4) is:

```
int x __attribute__((location( address )))
```

The NOINIT pragma may be used in conjunction with the LOCATION pragma to map variables to special memory locations; see Section 7.9.24.

7.9.22 The MUST_ITERATE Pragma

The MUST_ITERATE pragma specifies to the compiler certain properties of a loop. When you use this pragma, you are guaranteeing to the compiler that a loop executes a specific number of times or a number of times within a specified range.

Any time 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.

7.9.22.1 The MUST_ITERATE Pragma Syntax

The syntax of the pragma for C and C++ is:

#pragma MUST_ITERATE (*min*, *max*, *multiple* **)**

The C++ syntax for the corresponding attribute is as follows. No C attribute syntax is available.

[[TI::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.

The following example uses the must_iterate C++ attribute syntax:

```
void myFunc (int *a, int *b, int * restrict c, int n)
{
    ...
    [[TI::must_iterate(32, 1024, 16)]]
    for (int i = 0; i < n; i++)
    {
        c[i] = a[i] + b[i];
    }
    ...
}
```

7.9.22.2 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. The following example tells the compiler that the loop executes between 8 and 48 times and 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.

```
#pragma MUST_ITERATE(8, 48, 8)
for(i = 0; i < trip_count; i++)  { ...
```

You should consider using MUST_ITERATE for loops with complicated bounds. In the following example, the compiler would have to generate a divide function call to determine, at run time, the number of iterations performed.

```
for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

The compiler will not do the above. 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)  { ...
```

Typically, if the MUST_ITERATE pragma is used to optimize loop execution, a DINT instruction is prepended to the optimized code, the loop code is executed, and then an RINT instruction is executed when the loop is terminated.

7.9.23 The NMI_INTERRUPT Pragma

The NMI_INTERRUPT pragma enables you to handle non-maskable interrupts directly with C code. In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma NMI_INTERRUPT( func )
```

The syntax of the pragma in C++ is:

```
#pragma NMI_INTERRUPT
```

The code generated for the function will return via the NRP versus the IRP as for a function declared with the interrupt keyword or INTERRUPT pragma.

Except for _c_int00, which is the name reserved for the system reset interrupt for C programs, the name of the interrupt (function) does not need to conform to a naming convention.

7.9.24 The NOINIT and PERSISTENT Pragmas

Global and static variables are zero-initialized by default. However, in applications that use non-volatile memory, it may be desirable to have variables that are not initialized. Noinit variables are global or static variables that are not zero-initialized at startup or reset.

Variables can be declared as noinit or persistent using either pragmas or variable attributes. See Section 7.14.4 for information about using variable attributes in declarations.

Noinit and persistent variables behave identically with the exception of whether or not they are initialized at load time.

- The NOINIT pragma may be used only with uninitialized variables. It prevents such variables from being set to 0 during a reset. It may be used in conjunction with the LOCATION pragma to map variables to special memory locations, like memory-mapped registers, without generating unwanted writes.
- The PERSISTENT pragma may be used only with statically-initialized variables. It prevents such variables from being initialized during a reset. Persistent variables disable startup initialization; they are given an initial value when the code is loaded, but are never again initialized.

By default, noinit or persistent variables are placed in sections named `.TI.noinit` and `.TI.persistent` , respectively. The location of these sections is controlled by the linker command file. Typically .TI.persistent sections are placed in FRAM for devices that support FRAM and .TI.noinit sections are placed in RAM.

Note

When using these pragmas in non-volatile FRAM memory, the memory region could be protected against unintended writes through the device's Memory Protection Unit. Some devices have memory protection enabled by default. Please see the information about memory protection in the datasheet for your device. If the Memory Protection Unit is enabled, it first needs to be disabled before modifying the variables.

If you are using non-volatile RAM, you can define a persistent variable with an initial value of zero loaded into RAM. The program can increment that variable over time as a counter, and that count will not disappear if the device loses power and restarts, because the memory is non-volatile and the boot routines do not initialize it back to zero. For example:

```
#pragma PERSISTENT(x)
#pragma location = 0xC200   // memory address in RAM
int x = 0;
void main() {
    run_init();
    while (1) {
        run_actions(x);
        __delay_cycles(1000000);
        x++;
    }
}
```

The syntax of the pragmas in C is:

```
#pragma NOINIT ( x )
int x ;
```

```
#pragma PERSISTENT ( x )
int x =10;
```

The syntax of the pragmas in C++ is:

```
#pragma NOINIT
int x ;
```

```
#pragma PERSISTENT
int x =10;
```

The syntax of the GCC attributes is:

```
int x __attribute__((noinit));
```

```
int x __attribute__((persistent)) = 0;
```

7.9.25 The NOINLINE Pragma

The NOINLINE pragma can be placed before a statement to prevent any function calls made in that statement from being inlined. It has no effect on other calls to the same functions.

The syntax of the pragma in C/C++ is:

```
#pragma NOINLINE
```

For information about interactions between command-line options, pragmas, and keywords that affect inlining, see Section 3.11.

7.9.26 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 3.16 for details on entry and exit hooks.

7.9.27 The once Pragma

The once pragma tells the C preprocessor to ignore a #include directive if that header file has already been included. For example, this pragma may be used if header files contain definitions, such as struct definitions, that would cause a compilation error if they were executed more than once.

This pragma should be used at the beginning of a header file that should only be included once. For example:

```
// hdr.h
#pragma once
#warn You will only see this message one time
struct foo
{
    int member;
};
```

This pragma is not part of the C or C++ standard, but it is a widely-supported preprocessor directive. Note that this pragma does not protect against the inclusion of a header file with the same contents that has been copied to another directory.

7.9.28 The pack Pragma

The pack pragma can be used to control the alignment of fields within a class, struct, or union type. The syntax of the pragma in C/C++ can be any of the following.

```
#pragma pack ( n )
```

The above form of the pack pragma affects all class, struct, or union type declarations that follow this pragma in a file. It forces the maximum alignment of each field to be the value specified by *n*. Valid values for *n* are 1, 2, 4, 8, and 16 bytes.

```
#pragma pack ( push, n )
#pragma pack ( pop )
```

The above form of the pack pragma affects only class, struct, and union type declarations between push and pop directives. (A pop directive with no prior push results in a warning diagnostic from the compiler.) The maximum alignment of all fields declared is *n*. Valid values for *n* are 1, 2, 4, 8, and 16 bytes.

```
#pragma pack ( show )
```

The above form of the pack pragma sends a warning diagnostic to stderr to record the current state of the pack pragma stack. You can use this form while debugging.

For more about packed fields, see Section 7.14.5.

7.9.29 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* **)**

The C++ syntax for the corresponding attribute is as follows. No C attribute syntax is available. See Section 7.9.22.1 for an example that uses similar syntax.

[[TI::prob_iterate(*min*, *max* **)]]**

Where min and max are the minimum and maximum trip counts of the loop in the common case. The trip count is the number of times a loop iterates. Both arguments are optional.

For example, PROB_ITERATE could be applied to a loop that executes for eight iterations in the majority of cases (but sometimes may execute more or less than eight iterations):

```
#pragma PROB_ITERATE(8, 8)
```

If only the minimum expected trip count is known (say it is 5), the pragma would look like this:

```
#pragma PROB_ITERATE(5)
```

If only the maximum expected trip count is known (say it is 10), the pragma would look like this:

```
#pragma PROB_ITERATE(, 10)  /* Note the blank field for min */
```

7.9.30 The RETAIN Pragma

The RETAIN pragma can be applied to a code or data symbol.

It causes a .retain directive to be generated into the section that contains the definition of the symbol. The .retain directive indicates to the linker that the section is ineligible for removal during conditional linking. Therefore, regardless whether or not the section is referenced by another section in the application that is being compiled and linked, it will be included in the output file result of the link.

The RETAIN pragma has the same effect as using the `retain` function or variable attribute. See Section 7.14.2 and Section 7.14.4, respectively.

The syntax of the pragma in C is:

#pragma RETAIN (*symbol* **)**

The syntax of the pragma in C++ is:

#pragma RETAIN

7.9.31 The SET_CODE_SECTION and SET_DATA_SECTION Pragmas

These pragmas can be used to set the section for all declarations below the pragma.

The syntax of the pragmas in C/C++ is:

#pragma SET_CODE_SECTION (" *section name* **")**

#pragma SET_DATA_SECTION (" *section name* **")**

In the Setting Section With SET_DATA_SECTION Pragma example, x and y are put in the section mydata. To reset the current section to the default used by the compiler, a blank parameter should be passed to the pragma. An easy way to think of the pragma is that it is like applying the CODE_SECTION or DATA_SECTION pragma to all symbols below it.

Setting Section With SET_DATA_SECTION Pragma

```
#pragma SET_DATA_SECTION("mydata")
int x;
int y;
#pragma SET_DATA_SECTION()
```

The pragmas apply to both declarations and definitions. If applied to a declaration and not the definition, the pragma that is active at the declaration is used to set the section for that symbol. Here is an example:

Setting a Section With SET_CODE_SECTION Pragma

```
#pragma SET_CODE_SECTION("func1")
extern void func1();
#pragma SET_CODE_SECTION()
...
void func1() { ... }
```

In the Setting a Section With SET_CODE_SECTION Pragma example, func1 is placed in section func1. If conflicting sections are specified at the declaration and definition, a diagnostic is issued.

The current CODE_SECTION and DATA_SECTION pragmas and GCC attributes can be used to override the SET_CODE_SECTION and SET_DATA_SECTION pragmas. For example:

Overriding SET_DATA_SECTION Setting

```
#pragma DATA_SECTION(x, "x_data")
#pragma SET_DATA_SECTION("mydata")
int x;
int y;
#pragma SET_DATA_SECTION()
```

In the Overriding SET_DATA_SECTION Setting example, x is placed in x_data and y is placed in mydata. No diagnostic is issued for this case.

The pragmas work for both C and C++. In C++, the pragmas are ignored for templates and for implicitly created objects, such as implicit constructors and virtual function tables.

The SET_DATA_SECTION pragma takes precedence over the --gen_data_subsections=on option if it is used.

7.9.32 The STRUCT_ALIGN Pragma

The STRUCT_ALIGN pragma is similar to DATA_ALIGN, but it can be applied to a structure or union type or typedef. It 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.

Note that while the top-level object of a type (or a typedef of that type) will be aligned as requested, the type will not be padded to the alignment (as is usual for a struct), nor does the alignment propagate to derived types such as arrays and parent structs. If you want to pad a structure or union so that individual elements are also aligned and/or cause the alignment to apply to derived types, use the "aligned" type attribute described in Section 7.14.5.

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_typedef, 128)
```

Any use of STRUCT_ALIGN with a basic type (int, short, float) or a variable results in an error.

7.9.33 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. 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* **)**

The C++ syntax for the corresponding attribute is as follows. No C attribute syntax is available. See Section 7.9.22.1 for an example that uses similar syntax.

[[TI::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.

7.10 The _Pragma Operator

The C6000 C/C++ compiler supports the C99 preprocessor _Pragma() operator. This preprocessor operator is similar to #pragma directives. However, _Pragma can be used in preprocessing macros (#defines).

The syntax of the operator is:

_Pragma (" *string_literal* **");**

The argument *string_literal* is interpreted in the same way the tokens following a #pragma directive are processed. The string_literal must be enclosed in quotes. A quotation mark that is part of the string_literal must be preceded by a backward slash.

You can use the _Pragma operator to express #pragma directives in macros. For example, the DATA_SECTION syntax:

#pragma DATA_SECTION(*func* **,"** *section* **")**

Is represented by the _Pragma() operator syntax:

_Pragma ("DATA_SECTION(*func* **,\"** *section* **\")")**

The following code illustrates using _Pragma to specify the DATA_SECTION pragma in a macro:

```
...
#define EMIT_PRAGMA(x) _Pragma(#x)
#define COLLECT_DATA(var) EMIT_PRAGMA(DATA_SECTION(var,"mysection"))
COLLECT_DATA(x)
int x;
...
```

The EMIT_PRAGMA macro is needed to properly expand the quotes that are required to surround the section argument to the DATA_SECTION pragma.

7.11 Application Binary Interface

An Application Binary Interface (ABI) defines how functions that are written separately, and compiled or assembled separately can work together. This involves standardizing the data type representation, register conventions, and function structure and calling conventions. An ABI allows ABI-compliant object files to be linked together, regardless of their source, and allows the resulting executable to run on any system that supports that ABI. It defines linkname generation from C symbol names. It also defines the object file format and the debug format, along with documenting how the system is initialized. In the case of C++, it defines C++ name mangling and exception handling support.

The C6000 compiler and linker now support only the Embedded Application Binary Interface (EABI) ABI, which works only with object files that use the ELF object file format and the DWARF debug format. If you want support for the legacy COFF ABI, please use the C6000 v7.4.x Code Generation Tools and refer to SPRU187 and SPRU186 for documentation.

EABI uses the ELF object file format which enables supporting modern language features like early template instantiation and export inline functions support. TI-specific information on EABI mode is described in Section 8.9.2.

For low-level details about the C6000 EABI, see the *C6000 Embedded Application Binary Interface* (SPRAB89).

7.12 Object File Symbol Naming Conventions (Linknames)

Each externally visible identifier is assigned a unique symbol name to be used in the object file, a so-called *linkname*. This name is assigned by the compiler according to an algorithm which depends on the name, type, and source language of the symbol. This algorithm may add a prefix to the identifier (typically an underscore), and it may *mangle* the name.

User-defined symbols in C code and in assembly code are stored in the same namespace, which means you are responsible for making sure that your C identifiers do not collide with your assembly code identifiers. You may have identifiers that collide with assembly keywords (for instance, register names); in this case, the compiler automatically uses an escape sequence to prevent the collision. The compiler escapes the identifier with double parallel bars, which instructs the assembler not to treat the identifier as a keyword. You are responsible for making sure that C identifiers do not collide with user-defined assembly code identifiers.

Name mangling encodes the types of the parameters of a function in the linkname for a function. Name mangling only occurs for C++ functions which are not declared 'extern "C"'. Mangling allows function overloading, operator overloading, and type-safe linking. Be aware that the return value of the function is not encoded in the mangled name, as C++ functions cannot be overloaded based on the return value.

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 10 for more information.

The mangling algorithm follows that described in the Itanium C++ ABI (http://www.codesourcery.com/cxx-abi/abi.html).

`int foo(int i) { }` would be mangled "_Z3fooi"

7.13 Changing the ANSI/ISO C/C++ Language Mode

The language mode command-line options determine how the compiler interprets your source code. You specify one option to identify which language standard your code follows. You can also specify a separate option to specify how strictly the compiler should expect your code to conform to the standard.

Specify one of the following language options to control the language standard that the compiler expects the source to follow. The options are:

- ANSI/ISO C89 (--c89, default for C files)
- ANSI/ISO C99 (--c99, see Section 7.13.1.)
- ANSI/ISO C11 (--c11, see Section 7.13.2)
- ISO C++14 (--c++14 , used for all C++ files , see Section 7.2.)

Use one of the following options to specify how strictly the code conforms to the standard:

- Relaxed ANSI/ISO (--relaxed_ansi or -pr) This is the default.
- Strict ANSI/ISO (--strict_ansi or -ps)

The default is relaxed ANSI/ISO mode. Under relaxed ANSI/ISO mode, the compiler accepts language extensions that could potentially conflict with ANSI/ISO C/C++. Under strict ANSI mode, these language extensions are suppressed so that the compiler will accept all strictly conforming programs. (See Section 7.13.3.)

7.13.1 C99 Support (--c99)

The compiler supports the 1999 standard of C as standardized by the ISO. However, the following list of run-time functions and features are *not* implemented or fully supported:

- inttypes.h
  - wcstoimax() / wcstoumax()
- math.h
  - FP_ILOGB0 / FP_ILOGBNAN macros
  - MATH_ERRNO macro
  - copysign()
  - float_t / double_t types
  - math_errhandling()
  - signbit()
  - The following sets of C99 functions do not support the "long double" type. C89 math functions using float and long double types are supported. (Section numbers are from the C99 standard.)
    - 7.12.4: Trigonometric functions
    - 7.12.5: Hyperbolic functions
    - 7.12.6: Exponential and logarithmic functions
    - 7.12.7: Power and absolute value functions
    - 7.12.9: Nearest integer functions
    - 7.12.10: Remainder functions
  - expm1()
  - ilogb() / log1p() / logb()
  - scalbn() / scalbln()
  - cbrt()
  - hypot()
  - erf() / erfc()
  - lgamma() / tgamma()
  - nearbyint()
  - rint() / lrint() / llrint()
  - lround() / llround()
  - remainder() / remquo()
  - nan()
  - nextafter() / nexttoward()

  - fdim() / fmax() / fmin() / fma()
  - isgreater() / isgreaterequal() / isless() / islessequal() / islessgreater() / isunordered()
- stdio.h
  - The %e specifier may produce "-0" when "0" is expected by the standard
  - snprintf() does not properly pad with spaces when writing to a wide character array
- stdlib.h
  - vfscanf() / vscanf() / vsscanf() return value on floating point matching failure is incorrect
- wchar.h
  - getws() / fputws()
  - mbrlen()
  - mbsrtowcs()
  - wcscat()
  - wcschr()
  - wcscmp() / wcsncmp()
  - wcscpy() / wcsncpy()
  - wcsftime()
  - wcsrtombs()
  - wcsstr()
  - wcstok()
  - wcsxfrm()
  - Wide character print / scan functions
  - Wide character conversion functions

7.13.2 C11 Support (--c11)

The compiler supports the 2011 standard of C as standardized by the ISO. However, in addition to the list in Section 7.13.1, the following run-time functions and features are *not* implemented or fully supported in C11 mode:

- threads.h
- atomic operations

7.13.3 Strict ANSI Mode and Relaxed ANSI Mode (--strict_ansi and --relaxed_ansi)

Under relaxed ANSI/ISO mode (the default), the compiler accepts language extensions that could potentially conflict with a strictly conforming ANSI/ISO C/C++ program. Under strict ANSI mode, these language extensions are suppressed so that the compiler will accept all strictly conforming programs.

Use the --strict_ansi option when you know your program is a conforming program and it will not compile in relaxed mode. In this mode, language extensions that conflict with ANSI/ISO C/C++ are disabled and the compiler will emit error messages where the standard requires it to do so. Violations that are considered discretionary by the standard may be emitted as warnings instead.

Examples:

The following is strictly conforming C code, but will not be accepted by the compiler in the default relaxed mode. To get the compiler to accept this code, use strict ANSI mode. The compiler will suppress the interrupt keyword language exception, and interrupt may then be used as an identifier in the code.

```
int main()
{
    int interrupt = 0;
    return 0;
}
```

The following is not strictly conforming code. The compiler will not accept this code in strict ANSI mode. To get the compiler to accept it, use relaxed ANSI mode. The compiler will provide the interrupt keyword extension and will accept the code.

```
interrupt void isr(void);
int main()
{
    return 0;
}
```

The following code is accepted in all modes. The __interrupt keyword does not conflict with the ANSI/ISO C standard, so it is always available as a language extension.

```
__interrupt void isr(void);
int main()
{
    return 0;
}
```

The default mode is relaxed ANSI. This mode can be selected with the --relaxed_ansi (or -pr) option. Relaxed ANSI mode accepts the broadest range of programs. It accepts all TI language extensions, even those which conflict with ANSI/ISO, and ignores some ANSI/ISO violations for which the compiler can do something reasonable. Some GCC language extensions described in Section 7.14 may conflict with strict ANSI/ISO standards, but many do not conflict with the standards.

7.14 GNU and Clang Language Extensions

The GNU compiler collection (GCC) defines a number of language features not found in the ANSI/ISO C and C++ standards. The definition and examples of these extensions (for GCC version 4.7) can be found at the GNU web site, http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/C-Extensions.html#C-Extensions. Most of these extensions are also available for C++ source code.

The compiler also supports the following Clang macro extensions, which are described in the Clang 6 Documentation:

- __has_feature (up to tests described for Clang 3.5)
- __has_extension (up to tests described for Clang 3.5)
- __has_include
- __has_include_next
- __has_builtin (see Section 7.14.6)
- __has_attribute

7.14.1 Extensions

Most of the GCC language extensions are available in the TI compiler when compiling in relaxed ANSI mode (--relaxed_ansi).

The extensions that the TI compiler supports are listed in Table 7-6, which is based on the list of extensions found at the GNU web site. The shaded rows describe extensions that are not supported.

Table 7-6. GCC Language Extensions

Extensions	Descriptions
Statement expressions	Putting statements and declarations inside expressions (useful for creating smart 'safe' macros)
Local labels	Labels local to a statement expression
Labels as values	Pointers to labels and computed gotos
Nested functions	As in Algol and Pascal, lexical scoping of functions
Constructing calls	Dispatching a call to another function
Naming types(1)	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

Table 7-6. GCC Language Extensions (continued)

Extensions	Descriptions
Conditionals	Omitting the middle operand of a ?: expression
long long	Double long word integers and long long int type
Hex floats	Hexadecimal floating-point constants
Complex	Data types for complex number
Zero length	Zero-length arrays
Variadic macros	Macros with a variable number of arguments
Variable length	Arrays whose length is computed at run time
Empty structures	Structures with no members
Subscripting	Any array can be subscripted, even if it is not an lvalue.
Escaped newlines	Slightly looser rules for escaped newlines
Multi-line strings(1)	String literals with embedded newlines
Pointer arithmetic	Arithmetic on void pointers and function pointers
Initializers	Non-constant initializers
Compound literals	Compound literals give structures, unions, or arrays as values
Designated initializers	Labeling elements of initializers
Cast to union	Casting to union type from any member of the union
Case ranges	'Case 1 ... 9' and such
Mixed declarations	Mixing declarations and code
Function attributes	Declaring that functions have no side effects, or that they can never return
Attribute syntax	Formal syntax for attributes
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
Variable attributes	Specifying the attributes of variables
Type attributes	Specifying the attributes of types
Alignment	Inquiring about the alignment of a type or variable
Inline	Defining inline functions (as fast as macros)
Assembly labels	Specifying the assembler name to use for a C symbol
Extended asm	Assembler instructions with C operands
Constraints	Constraints for asm operands
Wrapper headers	Wrapper header files can include another version of the header file using #include_next
Alternate keywords	Header files can use __const__, __asm__, etc
Explicit reg vars	Defining variables residing in specified registers
Incomplete enum types	Define an enum tag without specifying its possible values
Function names	Printable strings which are the name of the current function
Return address	Getting the return or frame address of a function (limited support)
Other built-ins	Other built-in functions (see Section 7.14.6)
Vector extensions	Using vector operations (OpenCL syntax, see Section 7.15)
Target built-ins	Built-in functions specific to particular targets
Pragmas	Pragmas accepted by GCC
Unnamed fields	Unnamed struct/union fields within structs/unions
Thread-local	Per-thread variables
Binary constants	Binary constants using the '0b' prefix.

(1) Feature defined for GCC 3.0; definition and examples at http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/C-Extensions.html#C-Extensions

7.14.2 Function Attributes

The following GCC function attributes are supported:

- alias
- aligned
- always_inline
- calls
- const
- constructor
- deprecated
- format
- format_arg
- interrupt
- malloc
- naked
- noinline
- noreturn
- pure
- section
- unused
- used
- visibility
- warn_unused_result

The following additional TI-specific function attribute is supported:

- retain

For example, this function declaration uses the **alias** attribute to make "my_alias" a function alias for the "myFunc" function:

```
void my_alias() __attribute__((alias("myFunc")));
```

The **aligned** function attribute aligns the function using the specified alignment. The alignment must be a power of 2. This attribute has the same effect as the CODE_ALIGN pragma; see Section 7.9.2.

The **always_inline** function attribute has the same effect as the FUNC_ALWAYS_INLINE pragma. See Section 7.9.10

The **calls** attribute has the same effect as the CALLS pragma, which is described in Section 7.9.1.

The **format** attribute is applied to the declarations of printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf, scanf, fscanf, vfscanf, vscanf, vsscanf, and sscanf in stdio.h. Thus when GCC extensions are enabled, the data arguments of these functions are type checked against the format specifiers in the format string argument and warnings are issued when there is a mismatch. These warnings can be suppressed in the usual ways if they are not desired.

See Section 7.9.20 for more about using the **interrupt** function attribute.

The **malloc** attribute is applied to the declarations of malloc, calloc, realloc and memalign in stdlib.h.

The **naked** attribute identifies functions written as embedded assembly functions using `__asm` statements. The compiler does not generate prologue and epilog sequences for such functions. See Section 7.8.

The **noinline** function attribute has the same effect as the FUNC_CANNOT_INLINE pragma. See Section 7.9.11

The **retain** attribute has the same effect as the RETAIN pragma (Section 7.9.30). That is, the section that contains the function will not be omitted from conditionally linked output even if it is not referenced elsewhere in the application.

The **section** attribute when used on a function has the same effect as the CODE_SECTION pragma. See Section 7.9.3

7.14.3 For Loop Attributes

If you are using C++, there are several TI-specific attributes that can be applied to loops. No corresponding syntax is available in C. The following TI-specific attributes have the same function as their corresponding pragmas:

- TI::must_iterate
- TI::prob_iterate
- TI::unroll

See Section 7.9.22.1 for an example that uses a for loop attribute.

7.14.4 Variable Attributes

The following variable attributes are supported:

- aligned
- deprecated
- location
- mode
- noinit
- packed
- persistent
- retain
- section
- transparent_union
- unused
- used

The **aligned** attribute used on a variable has the same effect as the DATA_ALIGN pragma. See Section 7.9.4

The **location** attribute has the same effect as the LOCATION pragma. See Section 7.9.21. For example:

```
__attribute__((location(0x100))) extern struct PERIPH peripheral;
```

The **noinit** and **persistent** attributes apply to the ROM initialization model and allow an application to avoid initializing certain global variables during a reset. The alternative RAM initialization model initializes variables only when the image is loaded; no variables are initialized during a reset. See the "RAM Model vs. ROM Model" section and its subsections in the *TMS320C6000 Assembly Language Tools User's Guide*.

The **noinit** attribute can be used on uninitialized variables; it prevents those variables from being set to 0 during a reset. The **persistent** attribute can be used on initialized variables; it prevents those variables from being initialized during a reset. By default, variables marked noinit or persistent will be placed in sections named `.TI.noinit` and `.TI.persistent` , respectively. The location of these sections is controlled by the linker command file. Typically .TI.persistent sections are placed in FRAM for devices that support FRAM and .TI.noinit sections are placed in RAM. Also see Section 7.9.24.

The **packed** attribute may be applied to individual fields within a struct or union. The packed attribute for structure and union fields is available only when there is hardware support for unaligned accesses.

The **retain** attribute has the same effect as the RETAIN pragma (Section 7.9.30). That is, the section that contains the variable will not be omitted from conditionally linked output even if it is not referenced elsewhere in the application.

The **section** attribute when used on a variable has the same effect as the DATA_SECTION pragma. See Section 7.9.6

The **used** attribute is defined in GCC 4.2 (see http://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/Variable-Attributes.html#Variable-Attributes).

7.14.5 Type Attributes

The following type attributes are supported:

- aligned
- deprecated
- packed
- transparent_union
- unused
- visibility

Using the **aligned** type attribute causes the individual elements of a structure, union, or other type to be padded (as is usual for a struct) as needed to achieve the specified alignment for all elements. In addition, any derived types have the same alignment. For example:

```
struct __attribute__((aligned(32))) myStruct { char c1; int i; char c2; };
```

The **packed** attribute is supported for struct and union types. It is available only for target architectures that have hardware support for unaligned access if the --relaxed_ansi option is used.

Members of a packed structure are stored as closely to each other as possible, omitting additional bytes of padding usually added to preserve word-alignment. For example, assuming a word-size of 4 bytes ordinarily has 3 bytes of padding between members c1 and i, and another 3 bytes of trailing padding after member c2, leading to a total size of 12 bytes:

```
struct unpacked_struct { char c1; int i; char c2;};
```

However, the members of a packed struct are byte-aligned. Thus the following does not have any bytes of padding between or after members and totals 6 bytes:

```
struct __attribute__((__packed__)) packed_struct { char c1; int i; char c2; };
```

Subsequently, packed structures in an array are packed together without trailing padding between array elements.

Using "packed" on a bit-field overrides the EABI requirements for bit-fields. For *non-packed bit-fields*, the declared type of a bit-field is used for the container type. For *packed bit-fields*, the smallest integral type (except bool) is used, regardless of the declared type. For *non-packed volatile bit-fields*, the bit-field must be accessed using one access of the same size as the declared type. For *packed volatile bit-fields*, the access must be of the same size as the actual container type, and may not be the same as the declared type; additionally, the actual container might not be aligned, and might span more than one aligned container boundary, so accessing a packed volatile bit-field may require more than one memory access. This can also affect the overall size of the struct; for instance, if the struct contains only the bit-field, the struct might not be as large as the declared type of the bit-field. For *both packed and unpacked bit-fields*, bit-fields are bit-aligned and are packed together with adjacent bit-fields with no padding, are entirely contained within an integer container that is at least byte-aligned; and do not alter the alignment of adjacent non-bitfield struct members. See Section 8.2.2 for a description of bit-field layout.

The "packed" attribute can be applied only to the original definition of a structure or union type. It cannot be applied with a typedef to a non-packed structure that has already been defined, nor can it be applied to the declaration of a struct or union object. Therefore, any given structure or union type can only be packed or non-packed, and all objects of that type will inherit its packed or non-packed attribute.

The "packed" attribute is not applied recursively to structure types that are contained within a packed structure. Thus, in the following example the member s retains the same internal layout as in the first example above. There is no padding between c and s, so s falls on an unaligned boundary:

```
struct __attribute__((__packed__)) outer_packed_struct { char c; struct unpacked_struct s; };
```

It is illegal to implicitly or explicitly cast the address of a packed struct member as a pointer to any non-packed type except an unsigned char. In the following example, p1, p2, and the call to foo are all illegal.

```
void foo(int *param);
struct packed_struct ps;
int *p1 = &ps.i;
int *p2 = (int *)&ps.i;
foo(&ps.i);
```

However, it is legal to explicitly cast the address of a packed struct member as a pointer to an unsigned char:

```
unsigned char *pc = (unsigned char *)&ps.i;
```

The TI compiler also supports an **unpacked** attribute for an enumeration type to allow you to indicate that the representation is to be an integer type that is no smaller than int; in other words, it is not *packed*.

7.14.6 Built-In Functions

The following built-in functions are supported:

- __builtin_abs()
- __builtin_constant_p()
- __builtin_expect()
- __builtin_fabs()
- __builtin_fabsf()
- __builtin_frame_address()
- __builtin_labs()
- __builtin_memcpy()
- __builtin_return_address()

The __builtin_frame_address() function always returns zero unless the argument is a constant zero.

The __builtin_return_address() function always returns zero.

7.15 Operations and Functions for Vector Data Types

The C/C++ compiler supports the use of TI vector data types in C/C++ source files. Vector data types are described in Section 7.3.2. These vector data types are useful for parallel programming applications. You can enable support for vector data types by using the --vectypes compiler option.

The implementation of vector data types and operations follows the OpenCL C language specification closely. For a detailed description of OpenCL vector data types and operations, please see The OpenCL Specification version 1.2, which is available from the Khronos OpenCL Working Group.

Various types of operations can be performed on vectors. These include vector literals and concatenation (Section 7.15.1), unary and binary operators (Section 7.15.2), swizzle operators for component access (Section 7.15.3), and conversion operators (Section 7.15.4). In addition, several built-in functions (Section 7.15.7) are provided for working with vector types.

In addition to vector operations, printf() support is provided for outputting vector data. See Section 6.12.13 of The OpenCL Specification version 1.2 for details about formatting vector data types using printf(). For exceptions to the OpenCL specification regarding vector data types, see Section 7.15.6.

7.15.1 Vector Literals and Concatenation

You can specify the values to use for vector initialization or assignment using literals or scalar variables. When all of the components assigned to a vector are constants, the result is a vector literal. Otherwise, the vector's value is determined at run-time.

For example, the values assigned to vec_a and vec_b in the following declarations are vector literals and are known during compilation:

```
short4 vec_a = (short4)(1, 2, 3, 4);
float2 vec_b = (float2)(3.2, -2.3);
```

The following statements initialize all elements of a vector to the same value, which is 1 in this case.

```
ushort4 myushort4 = (ushort4)(1);
```

Shorter vectors can be concatenated together to form longer vectors. In the following example, two int variables are concatenated into an int2 variable. The value of myvec in the following function is not resolved until run-time:

```
void foo(int a, int b)
{
    int2 myvec = (int2)(a, b);
    ...
}
```

The following example concatenates two int2 variables into an int4 variable, which is passed to an external function:

```
extern void bar(int4 v4);
void foo(int a, int b)
{
    int2 myv2_a = (int2)(a, 1);
    int2 myv2_b = (int2)(b, 2);
    int4 myv4 = (int4)(myv2_a, myv2_b);
    bar(myv4);
}
```

7.15.2 Unary and Binary Operators for Vectors

When unary operators (such as negate: -) and binary operators (such as +) are applied to a vector, the operator is applied to each element in the vector. That is, each element in the resulting vector is the result of applying the operator to the corresponding elements in the source vector(s).

Table 7-7. Unary Operators Supported for Vector Types

Operator	Description
-	negate
~	bitwise complement
!	logical not (integer vectors only)

The following example declares an int4 vector called pos_i4 and initializes it to the values 1, 2, 3, and 4. It then uses the negate operator to initialize the values of another int4 vector, neg_i4, to the values -1, -2, -3, and -4.

```
int4 pos_i4 = (int4)(1, 2, 3, 4);
int4 neg_i4 = -pos_i4;
```

Table 7-8. Binary Operators Supported for Vector Types

Operator	Description
+, - , *, /	arithmetic operators (also supported for complex vectors)
=, +=, -=, *=, /=,	assignment operators
%	modulo operator (integer vectors only)
&, |, ^, <<, >>	bitwise operators
>, >=, ==, !=, <=, <	relational operators
++, --	increment / decrement operators (prefix and postfix; integer vectors only; also supported for the real portion of complex vectors)
&&, ||	logical operators (integer vectors only)

When binary operators are used with TI vector types, the element type and number of elements in each operand vector type must be the same. For arithmetic binary operators (e.g. +, -), the resulting type is equivalent to the operands' type.

Vector binary logical operators result in a vector type of the same number of elements as the vector operands with signed integer elements. For example, if an == operator compares two float4 types, the resulting type will be an int4. Comparing two double8 types results in a long8 type. A vector binary logical operator results in -1 (for true) or 0 (for false) in each result vector lane.

The following example uses the =, ++, and + operators on vectors of type int4. Assume that the iv4 argument initially contains (1, 2, 3, 4). On exit from foo(), iv4 will contain (3, 4, 5, 6).

```
void foo(int4 iv4)
{
    int4 local_iva = iv4++;                 /* local_iva = (1, 2, 3, 4) */
    int4 local_ivb = iv4++;                 /* local_ivb = (2, 3, 4, 5) */

    int4 local_ivc = local_iva + local_ivb; /* local_ivc = (3, 5, 7, 9) */
}
```

The arithmetic operators and increment / decrement operators can be used with complex vector types. The increment / decrement operators add or subtract by 1+0i.

The following example multiplies and divides complex vectors of type cfloat2. For details about the rules for complex multiplication and division, please see Annex G of the C99 C language specification.

```
void foo()
{
    cfloat2 va = (cfloat2) (1.0, -2.0, 3.0, -4.0);
    cfloat2 vb = (cfloat2) (4.0, -2.0, -4.0, 2.0);
    /* vc = (0.0, -10.0), (-4.0, 22.0) */
    cfloat2 vc = va * vb;
    /* vd = (0.4, -0.3), (-1.0, 0.5) */
    cfloat2 vd = va / vb;
    ...
}
```

On C64+ and C6740, the * and / operators in the previous example call a built-in function to perform the complex multiply and divide operations. On C6600, the compiler generates a CMPYSP instruction to carry out the complex multiply or divide operation.

7.15.3 Swizzle Operators for Vectors

The programming model implementation supports the following "swizzle" operators. These operators are used as suffixes to a variable name. They can be used on either side (left or right) of an assignment operator. When used on the left hand side of an assignment, each component must be uniquely identifiable.

.x, .y, .z, or .w Access an element of a vector whose length is <= 4.

```
char4 my_c4 = (char4)(1, 2, 3, 4);
char tmp = my_c4.y * my_c4.w;
/* ".y" accesses 2nd element ".w" accesses 4th element
 * tmp = 2 * 4 = 8; */
```

.s0, .s1, ..., .s9, .sa, ..., .sf Access one of up to 16 elements in a vector.

```
uchar16 ucvec16 = (uchar16)(1, 2, 3, 4, 5, 6, 7, 8,
                            9, 10, 11, 12, 13, 14, 15, 16 );
uchar8 ucvec8 = (uchar8)(2, 4, 6, 8, 10, 12, 14, 16);
int tmp = ucvec16.sa * ucvec8.s7;
/* ".sa" is 11th element of ucvec16
 * ".s7" is 8th element of ucvec8
 * tmp = 11 * 16 = 176;     */
```

.even, .odd Access the even or odd elements of a vector, where the zeroth element is even.

```
ushort4 usvec4 = (ushort4)(1, 2, 3, 4);
ushort2 usvecodd = usvec4.odd;   /* usvecodd = (ushort2)(2, 4); */
ushort2 usveceven = usvec4.even; /* usveceven = (ushort2)(1, 3); */
```

.hi, .lo Access the elements in the upper half of a vector with .hi or the elements in the lower half of a vector with .lo.

```
ushort8 usvec8 = (ushort8)(1, 2, 3, 4, 5, 6, 7, 8);
ushort4 usvechi = usvec8.hi; /* usvechi = (ushort4)(5, 6, 7, 8); */
ushort4 usveclo = usvec8.lo; /* usveclo = (ushort4)(1, 2, 3, 4); */
```

.r Access the real parts of each of the elements in a complex type vector.

```
cfloat2 cfa = (cfloat2)(1.0, -2.0, 3.0, -4.0);
float2 rfa = cfa.r;      /* rfa = (float2)(1.0, 3.0); */
```

.i Access the imaginary parts of each of the elements in a complex type vector.

```
cfloat2 cfa = (cfloat2)(1.0, -2.0, 3.0, -4.0);
float2 ifa = cfa.i;     /* ifa = (float2)(-2.0, -4.0); */
```

Swizzle operators can be combined to access a subset of the subset of elements. The result of the combination must be well-defined. For example, after the following code runs, usvec4 contains (1, 2, 5, 4).

```
ushort4 usvec4 = (ushort4)(1, 2, 3, 4);

usvec4.hi.even = 5;
```

7.15.4 Conversion Functions for Vectors

You cannot use standard type casting on vector data types. Instead, convert_<*destination type*>(<*source type*>) functions are provided to convert the elements of one vector type object into another vector type object. This is done on an element-by-element basis, and the source vector type and the destination vector type must be of the same length. That is, 4-element vectors can only be converted to other types of 4-element vectors.

The following example initializes a short2 vector using two ints concatenated to form an int2 vector:

```
void foo(int a, int b)
{
    short2 svec2 = convert_short2((int2)(a, b));
    ...
}
```

If the data stored by a vector element is outside the range of values that can be stored in the destination type, by default the value is truncated. However, if you add the _sat modifier (for "saturated") to the function name, values that are outside the range of the destination type are set to the maximum value of the destination type (or minimum for negative values outside the range). The _sat modifier cannot be used when converting an integer vector to a floating-point vector. In the following example, any values larger than 32,767 stored in an element of myint4 are set to as 32,767 in the corresponding element of myshort4.

```
int4 myint4;
short4 myshort4 = convert_short4_sat( myint4 );
```

Likewise, when converting between floating-point and integer vectors, one of the following modifiers can be added to the function name to specify how the floating-point values should be rounded:

- **_rte** — Round to nearest even integer
- **_rtz** — Round toward zero (default for converting floats to integers)
- **_rtp** — Round toward positive infinity
- **_rtn** — Round toward negative infinity

When converting an integer type to a float, rounding is not necessary. Rounding from a larger float type (double) to a smaller float type (float) does require rounding. The default rounding method for double to float conversions is _rte.

The following example converts the data stored in a float4 to an int4. It uses the _rtp modifier, so values are rounded up toward positive infinity:

```
float4 myfloat4;
int4 myint4 = convert_int4_rtp( myfloat4 );
```

The _sat modifier can be combined with a rounding modifier. The following example rounds floating values toward even and sets values greater than the maximum possible short value to the maximum value:

```
float4 myfloat4;
int4 myshort4 = convert_short4_sat_rte( myfloat4 );
```

If vector data types are enabled, you can also use the convert_<*type*>() functions for scalar (non-vector) types such as short and int. The result is the same as type casting the source type. Conversions between scalar types and vector types is not allowed, because the source and destination types must contain the same number of elements.

The convert_<*destination type*>() functions are not available for use with complex vector types.

7.15.5 Re-Interpretation Functions for Vectors

The as_<*destination type*>(<*source type object*>) functions are provided to re-interpret the original type of an object as another vector type. The source type and destination type must be the same size in number of bits. An error is returned if the sizes are different.

While arithmetic conversion is performed by the conversion functions described in the previous section, no arithmetic conversion is performed by the re-interpretation functions. For example, suppose a float value of 1.0 is re-interpreted as an int value. Since the float value of 1.0 is represented in hex as 0x3f800000, the value in the resulting int is 1,065,353,216.

The following example reinterprets a non-vector variable of the longlong type (64 bits) to a float2 vector (2 elements of 32 bits each). The least significant 32-bits of mylonglong are placed in fltvec2.s0 and the most significant 32-bits of mylonglong are placed in fltvec2.s1. No arithmetic conversion is performed.

```
extern longlong mylonglong;
float2 fltvec2 = as_float2(mylonglong);
```

If the sizes of the source and destination types are different, an error occurs.

If vector data types are enabled, you can also use the as_<*type*>() functions for scalar (non-vector) types. The types must have the same number of bits. The following example re-interprets a float value as an int value. Since the float value of 1.0 is represented in hex as 0x3f800000, the value in the resulting int is 1,065,353,216.

```
float myfloat = 1.0f;
myint = as_int(myfloat);
```

The as_<*destination type*>() functions are not available for use with complex vector types.

7.15.6 Using printf() with Vectors

In addition to vector operations, printf() support is provided for outputting vector data. See Section 6.12.13 of The OpenCL Specification version 1.2 for details about formatting vector data types using printf(). Note that the Texas Instruments C6000 implementation differs in the following ways from the OpenCL specification:

- **Return value.** The OpenCL specification indicates that printf() returns 0 or -1. To be consistent with the C99 specification for scalar printf(), the return value for vector printf() is the number of characters printed.
- **Length modifier for 64-bit integers.** However, the C6000 vector types do not include a long*n* or ulong*n* vector type, because both int and long are 32 bits. So, using a length modifier of %l (ell) would be inconsistent. Instead, use a length modifier of %ll (ell ell) for 64-bit long long and unsigned long long vector types.

That is, the `ll` length modifier specifies that a following d, i, o, u, x, or X conversion specifier applies to a longlong*n* or ulonglong*n* argument. Use the `l` length modifier for 64-bit double*n* arguments as described in the OpenCL specification.

The following example declares, initializes, and prints a vector of four 32-bit floating values, a vector of four 8-bit unsigned char values, and a vector of two 64-bit long long values.

```
float4 f4 = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
uchar4 uc = (uchar4)(0xFA, 0xFB, 0xFC, 0xFD);
longlong2 bigNums = (longlong2)(600000000000, -600000000000);

printf("f4 = %2.2v4hlf\n", f);
printf("uc = %#v4hhx\n", uc);
printf("bigNums = %+v2lld\n", bigNums);
```

The example prints the vector containing float values using the `%2.2v4hlf` format string to output values at least 2 digits wide with a precision of 2 (`2.2`) a vector of length 4 (`v4`) with a float*n* length modifier (`hl`) using the float type specifier (`f`).

The example prints the vector containing uchar values using the `%#v4hhx` format string to output a 0x prefix (`#`) followed by a vector of length 4 (`v4`) with a char*n* or uchar*n* length modifier (`hh`) using lowercase hexadecimal notation (`x`).

The example prints the vector containing long long values using the `%+v2lld` format string to output values with a + or - sign prefix (`+`) followed by a vector of length 2 (`v2`) with a longlong*n* or ulonglong*n* length modifier (`ll`) using decimal notation (`d`).

```
/* Output */
f4 = 1.00,2.00,3.00,4.00
uc = 0xfa,0xfb,0xfc,0xfd
bigNums = +600000000000,-600000000000
```

7.15.7 Built-In Vector Functions

Table 7-9 lists the built-in functions that can take vector arguments and return a vector result. Unless otherwise specified, the function is applied to each element in the vectors. That is, each element in the resulting vector is the result of applying the function to the corresponding elements in the source vectors. Unless otherwise specified, the vector types of all the arguments and the vector returned must be identical.

Table 7-9. Built-In Functions that Accept Vector Arguments

Function	Description	Vector Types Supported
__hadd(*x*, *y*)	Returns average using (*x* + *y*) >> 1. The intermediate sum does not modulo overflow. The vector types of *x*, *y*, and the vector returned must be identical.	short to short16 uchar to uchar16
__rhadd(*x*, *y*)	Returns average with rounding using (*x* + *y* + 1) >> 1. The intermediate sum does not modulo overflow.	short to short16 uchar to uchar16
__max(*x*, *y*)	Returns the greater value of *x* and *y* in each element of the returned vector.	short to short16 uchar to uchar16
__min(*x*, *y*)	Returns the lesser value of *x* and *y* in each element of the returned vector.	short to short16 uchar to uchar16
__add_sat(*x*, *y*)	Returns *x* + *y* and saturates the result. That is, if there is an overflow, the maximum value in range for the type is returned.	short to short16 ushort to ushort16 uchar to uchar16 int to int16
__sub_sat(*x*, *y*)	Returns *x* - *y* and saturates the result. That is, if there is an overflow, the maximum value in range for the type is returned.	short to short16 int to int16
__abs_diff(*x*, *y*)	Returns the absolute difference using | *x* - *y* | without overflow.	uchar to uchar16
__abs(*x*)	Returns the absolute value using | *x* |	short to short16

Table 7-9. Built-In Functions that Accept Vector Arguments (continued)

Function	Description	Vector Types Supported
__popcount(*x*)	Returns the number of non-zero bits in *x*.	uchar to uchar16
__mpy_ext(*x*, *y*)	Extended precision multiplication.	short to short16 ushort to ushort16 char to char16 uchar to uchar16
__mpy_fx_ext(*x*, *y*)	Fixed point multiplication.	short to short16
__conj_cmpy(*x*, *y*)	Conjugate complex multiplication.	cfloat to cfloat8
__cmpy_ext(*x*, *y*)	Extended precision complex multiplication.	cshort to cshort8
__cmpyr_fx(*x*, *y*)	Fixed point complex multiplication with rounding.	cshort to cshort8
__cmpy_fx(*x*, *y*)	Fixed point complex multiplication.	cshort to cshort8 cint to cint8
__conj_cmpy_fx(*x*, *y*)	Fixed point conjugate complex multiplication. (C6600 only)	cint to cint8
__crot90(*x*)	Rotate by 90 degrees. (C6600 only)	cshort to cshort8
__crot270(*x*)	Rotate by 270 degrees. (C6600 only)	cshort to cshort8
__dot_ext(*x*, *y*)	Extended precision dot product using *x* • *y*.	short2 factors give an int result char4 factors give an int result uchar4 factors give a uint result short4 factors give an int result (C6600 only) short4 and ushort4 factors give an int result (C6600 only)
__dot_extll(*x*, *y*)	Extended precision dot product using *x* • *y*.	short2 factors give a longlong result short4 factors give a longlong result (C6600 only) short4 and ushort4 factors give a longlong result (C6600 only)
__dot_fx(*x*, *y*)	Fixed point dot product using *x* • *y*.	short2 • ushort2 = int
__gmpy(*x*, *y*)	Galois field multiplication.	uchar4
__ddot_ext(*x*, *y*)	Extended precision 2-way dot product.	short2 • char4 = int4 short8 • short8 = int2 (C6600 only) short8 • ushort8 = int2 (C6600 only)
__mpy_fx(*x*, *y*)	Fixed point multiplication.	short2 * int = int2 int4 * int4 = int4 (C6600 only)
__apply_sign(*x*, *y*)	Uses the sign bit of x to determine whether to multiply the four 16-bit values in y by 1 or -1. Yields four signed 16-bit results. This function is an alias for the _dapys2() intrinsic. (C6600 only)	short4
__cmpy_conj_ext(*x*, *y*)	Extended precision complex multiplication conjugate. (C6600 only)	cshort2 * cshort2 = cint2
__cmpy_conj_fx(*x*, *y*)	Fixed point complex multiplication conjugate. (C6600 only)	cshort2 * cshort2 = cshort2
__cmatmpy_ext(*x*, *y*)	Extended precision complex matrix multiplication. (C6600 only)	cshort2 * cshort4 = cint2
__conj_cmatmpy_ext(*x*, *y*)	Extended precision complex matrix multiplication conjugate. (C6600 only)	cshort2 * cshort4 = cint2
__cmatmpy_fx(*x*, *y*)	Fixed point complex matrix multiplication. (C6600 only)	cshort2 * cshort4 = cshort2
__conj_cmatmpy_fx(*x*, *y*)	Fixed point complex matrix multiplication conjugate. (C6600 only)	cshort2 * cshort4 = cshort2

Prototypes for all the supported vector built-in functions are listed in the "c6x_vec.h" header file, which is located in the "include" sub-directory of your Code Generation Tools installation. Please see the "c6x_vec.h" for a complete list of the vector built-in functions.

The following example vbif_ex.c file uses the __add_sat() and __sub_sat() built-in functions with vectors:

```
#include <stdio.h>
#include <c6x_vec.h>
void print_short4(char *s, short4 v)
{
    printf("%s", s);
    printf(" <%d, %d, %d, %d>\n", v.x, v.y, v.z, v.w);
}
int main()
{
    short4 va = (short4) (1, 2, 3, -32766);
    short4 vb = (short4) (5, 32767, -13, 17);
    short4 vc = va + vb;
    short4 vd = va - vb;
    short4 ve = __add_sat(va, vb);
    short4 vf = __sub_sat(va, vb);
    print_short4("va=", va);
    print_short4("vb=", vb);
    print_short4("vc=(va+vb)=", vc);
    print_short4("vd=(va-vb)=", vd);
    print_short4("ve=__add_sat(va,vb)=", ve);
    print_short4("vf=__sub_sat(va,vb)=", vf);
    return 0;
}
```

Compile the example as follows:

```
%> cl6x -mv6400 --vectypes -o1 -k vbif_ex.c -z -o vbif_ex.out -llnk.cmd
```

Note that the lnk.cmd file contains a reference to rts6400.lib. The rts6400.libs library contains c6x_veclib.obj, which defines the built-in functions, __add_sat() and __sub_sat().

Running this example produces the following output:

```
va= <1, 2, 3, -32766>
vb= <5, 32767, -13, 17>
vc=(va+vb)= <6, -32767, -10, -32749>
vd=(va-vb)= <-4, -32765, 16, 32753>
ve=__add_sat(va,vb)= <6, 32767, -10, -32749>
vf=__sub_sat(va,vb)= <-4, -32765, 16, -32768>
```

This page intentionally left blank.

Chapter 8

Run-Time Environment

This chapter describes the TMS320C6000 C/C++ run-time environment. To ensure successful execution of C/C++ programs, it is critical that all run-time code maintain this environment. It is also important to follow the guidelines in this chapter if you write assembly language functions that interface with C/C++ code.

8.1 Memory Model

The C6000 compiler treats memory as a single linear block that is partitioned into subblocks of code and data. Each subblock of code or data generated by a C program is placed in its own continuous memory space. The compiler assumes that a full 32-bit address space is available in target memory.

Note

The Linker Defines the Memory Map

The linker, not the compiler, defines the memory map and allocates code and data into target memory. The compiler assumes nothing about the types of memory available, about any locations not available for code or data (holes), or about any locations reserved for I/O or control purposes. The compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces. For example, you can use the linker to allocate global variables into on-chip RAM or to allocate executable code into external ROM. You can allocate each block of code or data individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although you can access physical memory locations with C/C++ pointer types).

8.1.1 Sections

The compiler produces relocatable blocks of code and data called *sections*, which are allocated in memory in a variety of ways to conform to a various system configurations. For information about sections and allocating them, see the introductory object file information in the *TMS320C6000 Assembly Language Tools User's Guide*.

There are two basic types of sections:

- **Initialized sections** contain data or executable code. Initialized sections are usually read-only; exceptions are noted below. The C/C++ compiler creates the following initialized sections:
  - The **.args section** contains the command argument for a host-based loader. See the --arg_size option.
  - The **.binit section** contains boot time copy tables. For details on BINIT, see the *TMS320C6000 Assembly Language Tools User's Guide*.
  - The **.cinit section** is created only if you are using the --rom_model option. It contains tables for explicitly initialized global and static variables.
  - The **.init_array section** contains the table for calling global constructors.
  - The **.ovly section** contains copy tables for unions in which different sections have the same run address.
  - The **.c6xabi.exidx section** contains the index table for exception handling. The **.c6xabi.extab section** contains stack unwinding instructions for exception handling. See the --exceptions option for details.
  - The **.** *name* **.load section** contains the compressed image of section *name*.See the *TMS320C6000 Assembly Language Tools User's Guide* for information on copy tables.
  - The **.ppdata section** contains data tables for compiler-based profiling. See the --gen_profile_info option for details. This section is writable.
  - The **.ppinfo section** contains correlation tables for compiler-based profiling. See the --gen_profile_info option for details.
  - The **.const section** contains string literals, floating-point constants, and data defined with the C/C++ qualifiers *far* and *const* (provided the constant is not also defined as *volatile* or one of the exceptions described in Section 7.5.2). String literals are placed in the .const:.string subsection to enable greater link-time placement control.
  - The **.fardata section** reserves space for non-const, initialized far global and static variables. This section is writable.
  - The **.neardata section** reserves space for non-const, initialized near global and static variables. This section is writable.
  - The **.rodata section** reserves space for const near global and static variables.
  - The **.switch section** contains jump tables for large switch statements.
  - The **.text section** contains all the executable code and compiler-generated constants. This section is usually read-only.
  - The **.TI.crctab section** contains CRC checking tables.

- **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 uninitialized global and static variables. Uninitialized variables that are also unused are usually created as common symbols (unless you specify --common=off) instead of being placed in .bss so that they can be excluded from the resulting application.
  - The **.far section** reserves space for global and static variables that are declared far.
  - The **.stack section** reserves memory for the system stack.
  - The **.sysmem section** reserves space for dynamic memory allocation. This space is used by dynamic memory allocation routines, such as malloc(), calloc(), realloc(), or new(). 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 code sections, the initialized and uninitialized sections cannot be allocated into internal program memory.

The assembler creates the default sections .text, .bss, and .data. You can instruct the compiler to create additional sections by using the CODE_SECTION and DATA_SECTION pragmas (see Section 7.9.3 and Section 7.9.6).

8.1.2 C/C++ System Stack

The C/C++ compiler uses a stack to:

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

The run-time stack grows from the high addresses to the low addresses. The compiler uses the B15 register to manage this stack. B15 is the *stack pointer* (SP), which points to the next unused location on the stack.

The linker sets the stack size, creates a global symbol, __TI_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 the linker description chapter in the *TMS320C6000 Assembly Language Tools User's Guide*.

At system initialization, SP is set to the first 8-byte (64-bit) 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 8.4.

Note

Unaligned SP Can Cause Application Crash: The HWI dispatcher uses SP during an interrupt call regardless of SP alignment. Therefore, SP must never be misaligned, even for 1 cycle.

Note

Stack Overflow: The compiler provides no means to check for stack overflow during compilation or at run time. A stack overflow disrupts the run-time environment, causing your program to fail. Be sure to allow enough space for the stack to grow. You can use the --entry_hook option to add code to the beginning of each function to check for stack overflow; see Section 3.16.

8.1.3 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, __TI_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 the linker description chapter in the *TMS320C6000 Assembly Language Tools User's Guide*.

If you use any C I/O function, the RTS library allocates an I/O buffer for each file you access. This buffer will be a bit larger than BUFSIZ, which is defined in stdio.h and defaults to 256. Make sure you allocate a heap large enough for these buffers or use setvbuf to change the buffer to a statically-allocated buffer.

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 size may be limited only by the amount of 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));
```

When allocating from a heap, make sure the size of the heap is large enough for the allocation. This is particularly important when allocating variable-length arrays.

8.1.4 Data Memory Models

Several options extend the C6000 data addressing model.

8.1.4.1 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 7.5.5.

8.1.4.2 DP-Relative Vs. Absolute Addressing

The compiler uses DP-relative addressing for near (.bss) data. It uses absolute addressing for all other (far) data.

8.1.4.3 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.

8.1.5 Trampoline Generation for Function Calls

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.

8.1.6 Position Independent Data

Near global and static data are stored in the .bss section. All near data for a program must fit within 32K bytes of memory. This limit comes from the addressing mode used to access near data, which is limited to a 15-bit unsigned offset from DP (B14), 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.

8.2 Object Representation

This section explains how various data objects are sized, aligned, and accessed.

8.2.1 Data Type Storage

For general information about data types, see Section 7.3. Table 8-1 lists register and memory storage for various data types:

Table 8-1. Data Representation in Registers and Memory

Data Type	Register Storage	Memory Storage
char	Bits 0-7 of register	8 bits aligned to 8-bit boundary
unsigned char	Bits 0-7 of register	8 bits aligned to 8-bit boundary
short	Bits 0-15 of register	16 bits aligned to 16-bit boundary
unsigned short	Bits 0-15 of register	16 bits aligned to 16-bit boundary
int	Entire register	32 bits aligned to 32-bit boundary
unsigned int	Entire register	32 bits aligned to 32-bit boundary
enum (1)	Entire register or even/odd register pair	32 bits aligned to 32-bit boundary or 64 bits aligned to 64-bit boundary
float	Entire register	32 bits aligned to 32-bit boundary
float complex (4)	Even/odd register pair	64 bits aligned to 32-bit boundary
long	Entire register	32 bits aligned to 32-bit boundary
unsigned long	Entire register	32 bits aligned to 32-bit boundary
__int40_t	Even/odd register pair	64 bits aligned to 64-bit boundary
unsigned __int40_t	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
double complex (C6600 only)(4)	Register quad(5)	128 bits aligned to 64-bit boundary
long double	Even/odd register pair	64 bits aligned to 64-bit boundary
long double complex (C6600 only)(4)	Register quad(5)	128 bits aligned to 64-bit boundary
__x128_t (C6600 only)(2)	Register quad(5)	128-bits aligned to 64-bit boundary
struct	Members are stored as their individual types require.	Storage is a multiple of the alignment to the 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.(3) 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	32 bits aligned to 32-bit boundary

(1) For details about the size of an enum type, see Section 7.3.1.
(2) For details on the __x128_t container type, see Section 8.6.7.
(3) Static scope arrays are aligned to a 64-bit boundary.
(4) To use complex data types, you must include the <complex.h> header file. See Section 7.5.1 for more about how complex data types are stored.
(5) Register quads are supports for C66000 only.

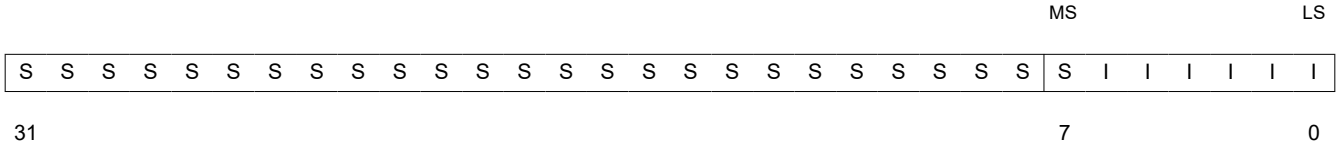
8.2.1.1 char and short Data Types (signed and unsigned)

The char and unsigned char data types are stored in memory as a single byte and are loaded to and stored from bits 0-7 of a register (see Figure 8-1). Objects defined as short or unsigned short are stored in memory as two bytes at a halfword (2 byte) aligned address and they are loaded to and stored from bits 0-15 of a register (see Figure 8-1).

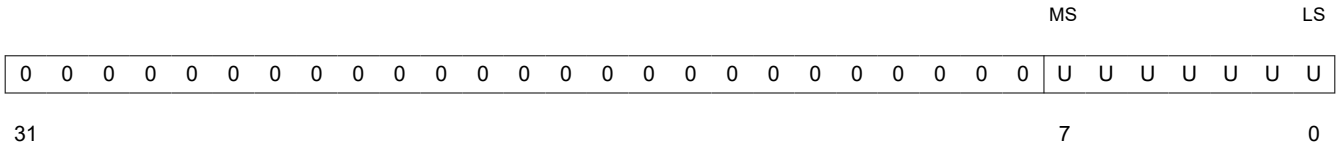
In big-endian mode, 2-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 8-15 of the register and moving the second byte of memory to bits 0-7. In little-endian mode, 2-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 0-7 of the register and moving the second byte of memory to bits 8-15.

Figure 8-1. Char and Short Data Storage Format

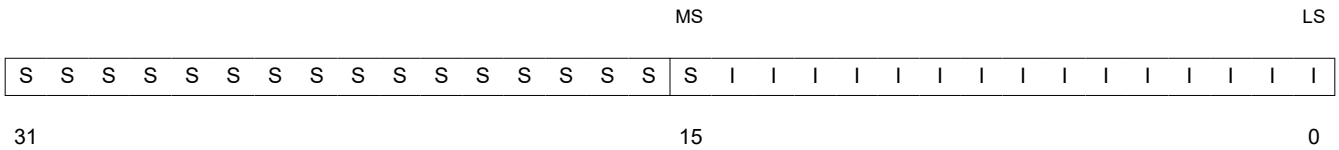
Signed 8-bit char



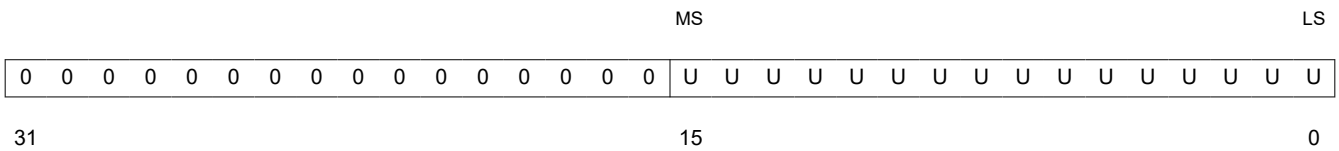
Unsigned 8-bit char



Signed 16-bit short



Unsigned 16-bit short



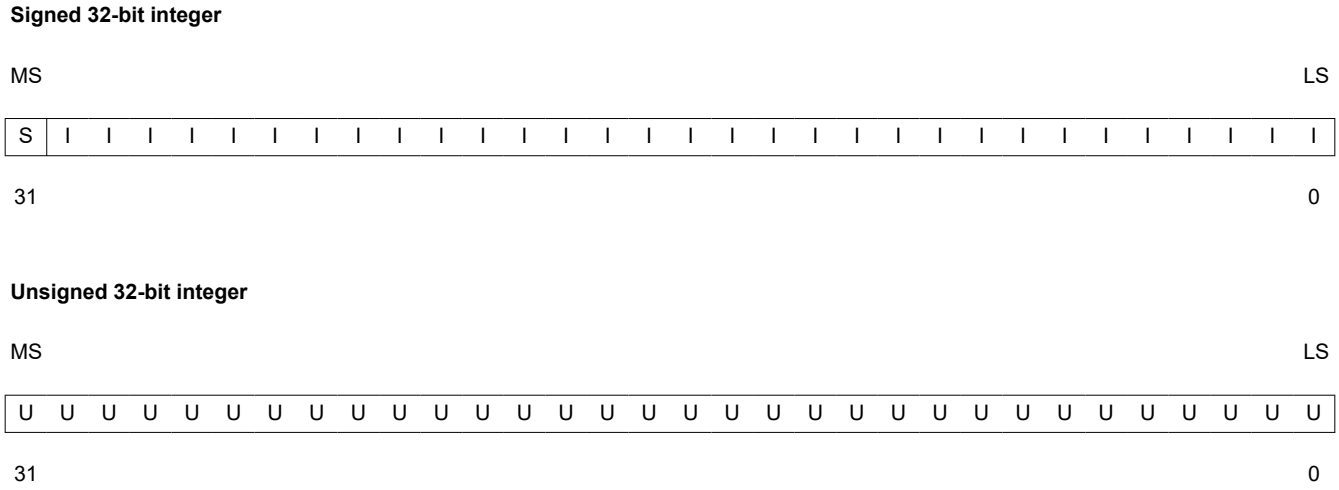
LEGEND: S = sign, I = signed integer, U = unsigned integer, MS = most significant, LS = least significant

8.2.1.2 enum, int, and long Data Types (signed and unsigned)

The int and unsigned int data types are stored in memory as 32-bit objects (see Figure 8-2). Objects of these types are loaded to and stored from bits 0-31 of a register. In big-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 24-31 of the register, moving the second byte of memory to bits 16-23, moving the third byte to bits 8-15, and moving the fourth byte to bits 0-7. In little-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 0-7 of the register, moving the second byte to bits 8-15, moving the third byte to bits 16-23, and moving the fourth byte to bits 24-31.

For details about the size of an enum type, see Section 7.3.1.

Figure 8-2. 32-Bit Data Storage Format

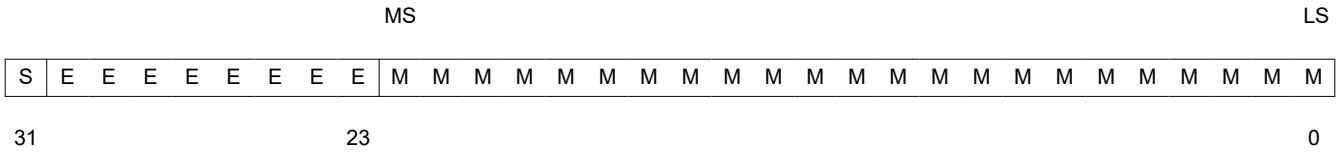


LEGEND: S = sign, U = unsigned integer, I = signed integer, MS = most significant, LS = least significant

8.2.1.3 float Data Type

The float data type is stored in memory as 32-bit objects (see Figure 8-3). Objects defined as float are loaded to and stored from bits 0-31 of a register. In big-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 24-31 of the register, moving the second byte of memory to bits 16-23, moving the third byte to bits 8-15, and moving the fourth byte to bits 0-7. In little-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 0-7 of the register, moving the second byte to bits 8-15, moving the third byte to bits 16-23, and moving the fourth byte to bits 24-31.

Figure 8-3. Single-Precision Floating-Point Char Data Storage Format

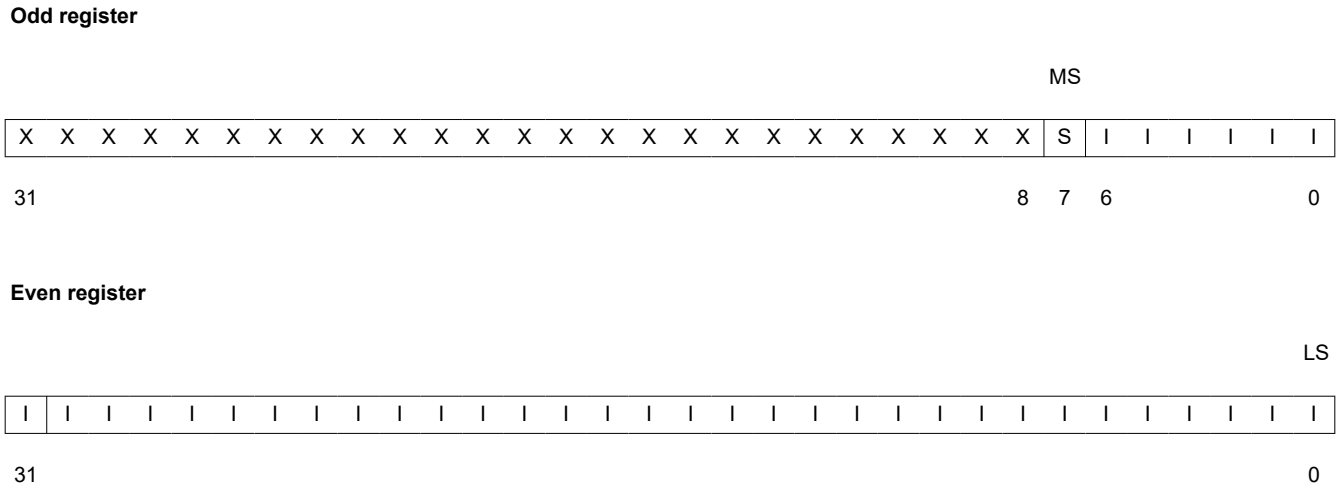


LEGEND: S = sign, M = mantissa, E = exponent, MS = most significant, LS = least significant

8.2.1.4 The __int40_t Data Type (signed and unsigned)

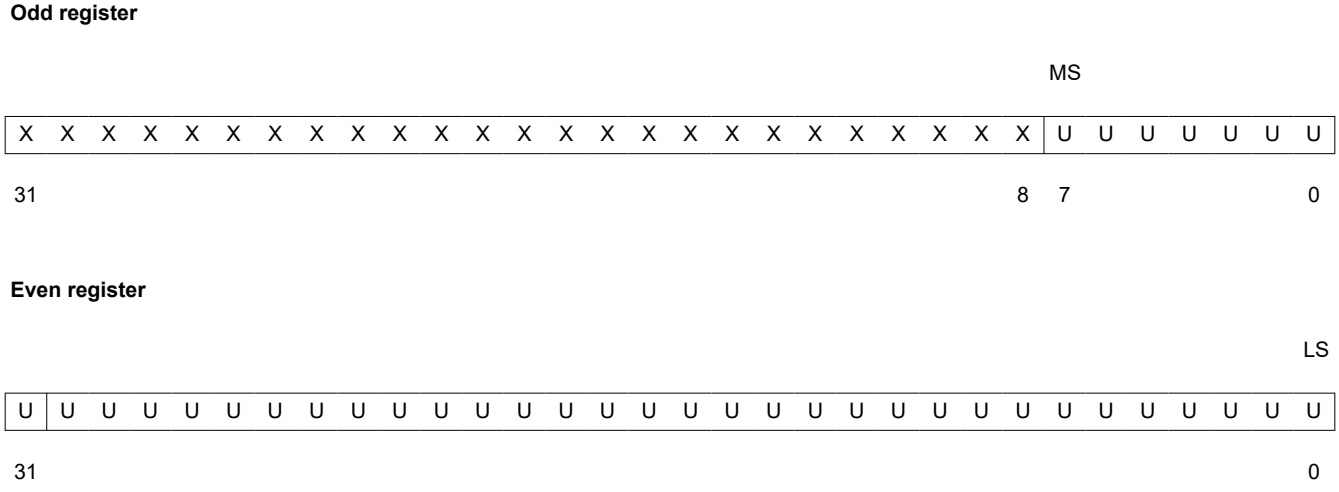
The __int40_t data type is stored in an odd/even pair of registers (see Figure 8-4) and is always referenced as a pair in the format of odd register:even register (for example, A1:A0). In little-endian mode, the lower address is loaded into the even register and the higher address is loaded into the odd register; if data is loaded from location 0, then the byte at 0 is the lowest byte of the even register. In big-endian mode, the higher address is loaded into the even register and the lower address is loaded into the odd register; if data is loaded from location 0, then the byte at 0 is the highest byte of the odd register but is ignored.

Figure 8-4. 40-Bit Data Storage Format Signed __int40_t



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

Figure 8-5. Unsigned 40-bit __int40_t

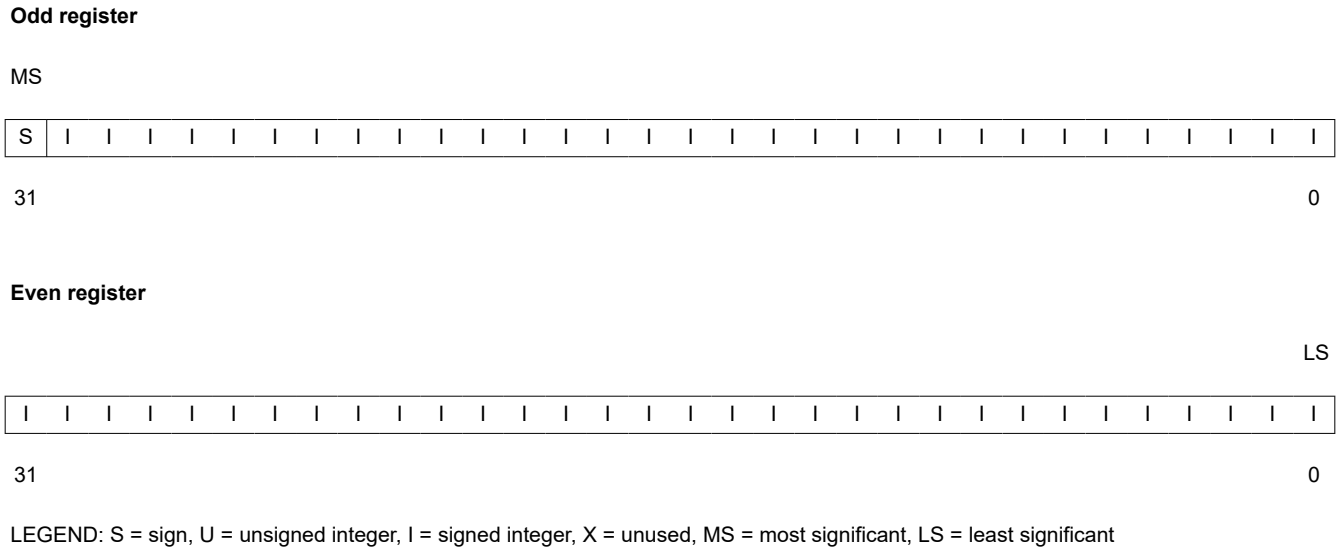


LEGEND: S = sign, U = unsigned integer, I = signed integer, X = unused, MS = most significant, LS = least significant

8.2.1.5 long long Data Types (signed and unsigned)

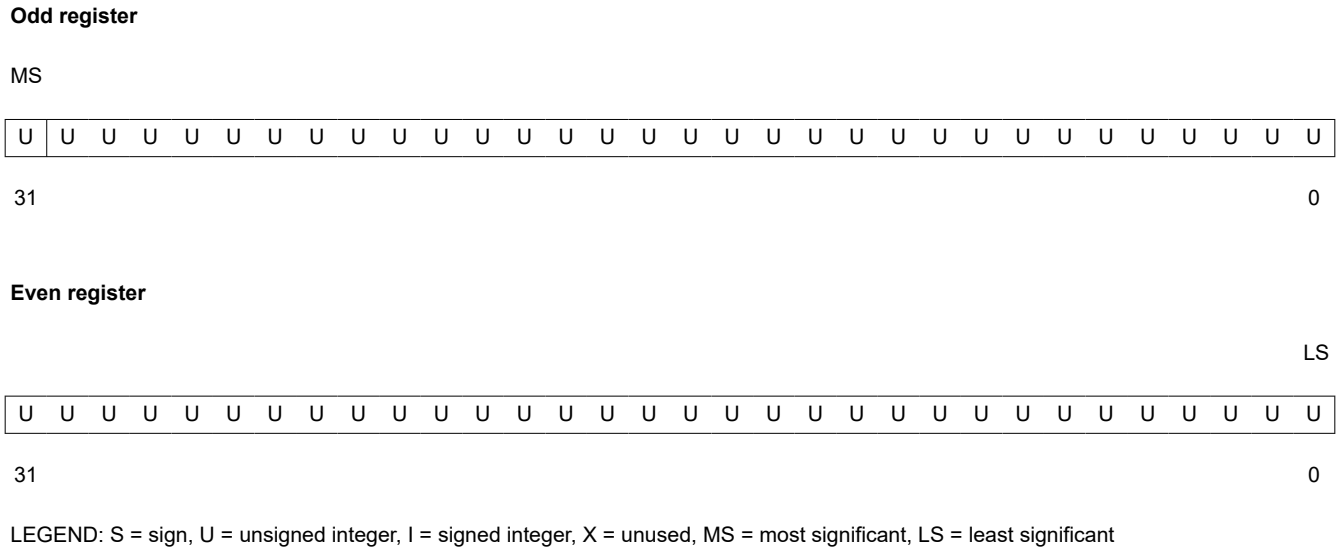
Long long and unsigned long long data types are stored in an odd/even pair of registers (see Figure 8-6) and are always referenced as a pair in the format of odd register:even register (for example, A1:A0). In little-endian mode, the lower address is loaded into the even register and the higher address is loaded into the odd register; if data is loaded from location 0, then the byte at 0 is the lowest byte of the even register. In big-endian mode, the higher address is loaded into the even register and the lower address is loaded into the odd register; if data is loaded from location 0, then the byte at 0 is the highest byte of the odd register.

Figure 8-6. 64-Bit Data Storage Format Signed 64-bit long



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

Figure 8-7. Unsigned 64-bit long

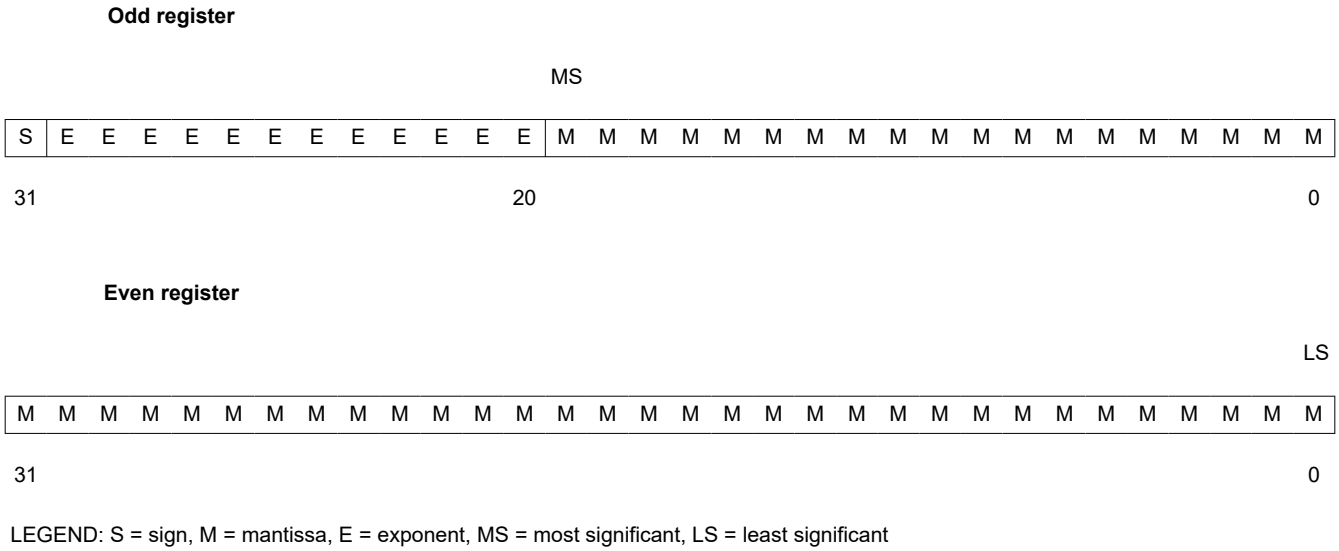


LEGEND: S = sign, U = unsigned integer, I = signed integer, X = unused, MS = most significant, LS = least significant

8.2.1.6 double and long double Data Types

Double and long double data types are stored in an odd/even pair of registers (see Figure 8-8) and can only exist in a register in one format: as a pair in the format of odd register:even register (for example, A1:A0). The odd memory word contains the sign bit, exponent, and the most significant part of the mantissa. The even memory word contains the least significant part of the mantissa. In little-endian mode, the lower address is loaded into the even register and the higher address is loaded into the odd register. In big-endian mode, the higher address is loaded into the even register and the lower address is loaded into the odd register. In little-endian mode, if code is loaded from location 0, then the byte at 0 is the lowest byte of the even register. In big-endian mode, if code is loaded from location 0, then the byte at 0 is the highest byte of the odd register.

Figure 8-8. Double-Precision Floating-Point Data Storage Format



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

8.2.1.7 Pointer to Data Member Types

Pointer to data member objects are stored in memory like an unsigned int (32 bit) integral type. Its value is the byte offset to the data member in the class, plus 1. The zero value is reserved to represent the NULL pointer.

8.2.1.8 Pointer to Member Function Types

Pointer to member function objects have a layout equivalent to:

```
struct __mptr {
   __vptp f;
   ptrdiff_t d;
};
```

where _vptp is the following union:

```
union {
   void (*f) ();
   int 0;
}
```

The parameter f is the pointer to the member function if it is nonvirtual. The 0 is the offset to the virtual function pointer within the class object. The parameter d is the offset to be added to the beginning of the class object for this pointer.

8.2.1.9 Structures and Arrays

A struct is aligned to a boundary required by the member it contains with the strictest alignment requirement. For example, if the largest alignment required by a member of the struct is 16-bit alignment (for example, a short),

then the entire struct is aligned to a 16-bit boundary. If the struct contains a type that requires 64-bit alignment (such as a double or long long), then the struct is aligned to a 64-bit boundary.

If a struct member is itself a struct, the size and alignment of the inner struct must be determined before the size and alignment of the outer struct may be determined.

Members of structs have sizes and alignments equal to those they would have as independent objects, unless the packed attribute is used. An array member of a struct is aligned to the alignment of its element type; this may differ from the alignment the element would have if it were an independent top-level (static) object.

Structs always have size equal to a multiple of the struct alignment. This sometimes requires padding after the last member to round the size up to a multiple of the struct alignment. The size of a structure includes any necessary padding between members. For example, if the largest member of a struct is of type float, the size of the struct will be a multiple of 32 bits.

Static scope arrays (sometimes called top-level arrays) are aligned on an 8-byte (64-bit) boundary.

8.2.2 Bit Fields

Bit fields are the only objects that are packed within a byte. That is, two bit fields can be stored in the same byte. Bit fields can range in size from 1 to 64 bits in C or larger in C++.

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.

The size, alignment, and type of bit fields adhere to these rules:

- Bit fields up to long long are supported.
- Bit fields are treated as the declared signed or unsigned type.
- The size and alignment of the struct containing a bit field depends on the declared type of the bit field. For example, consider the struct:

```
struct st
{
   int a:4
};
```

  This struct uses up 4 bytes and is aligned at 4 bytes.
- Unnamed bit fields do affect the alignment of the struct or union. For example, consider the struct:

```
struct st
{
   char a:4;
   int :22;
};
```

  This struct uses 4 bytes and is aligned at a 4-byte boundary.
- Bit fields declared volatile are accessed according to the bit field's declared type. A volatile bit field reference generates exactly one reference to its storage; multiple volatile bit field accesses are not merged.

Figure 8-9 illustrates bit-field packing, using the following bit field definitions:

```
struct{
int A:7
int B:10
int C:3
int D:2
int E:9
}x;
```

A0 represents the least significant bit of the field A; A1 represents the next least significant bit, etc. Again, storage of bit fields in memory is done with a byte-by-byte, rather than bit-by-bit, transfer.

Figure 8-9. 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	C	C	C	D	D	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	A	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

8.2.3 Character String Constants

In C, a character string constant is used in one of the following ways:

- To initialize an array of characters. For example:

```
char s[] = "abc";
```

  When a string is used as an initializer, it is simply treated as an initialized array; each character is a separate initializer. For more information about initialization, see Section 8.9.
- In an expression. For example:

```
strcpy (s, "abc");
```

  When a string is used in an expression, the string itself is defined in the .const:string section with the .string assembler directive, along with a unique label that points to the string; the terminating 0 byte is explicitly

added by the compiler. For example, the following lines define the string abc, and the terminating 0 byte (the label SL5 points to the string):

```
        .sect   ".const:string"
$C$SL5: .string "abc",0
```

String labels have the form CSLn, where C is the compiler-generated symbol prefix and *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 CSL*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! undefined behavior */
```

8.3 Register Conventions

Strict conventions associate specific registers with specific operations in the C/C++ environment. If you plan to interface an assembly language routine to a C/C++ program, you must understand and follow these register conventions.

The register conventions dictate how the compiler uses registers and how values are preserved across function calls. Table 8-2 summarizes how the compiler uses the TMS320C6000 registers.

The registers in Table 8-2 are available to the compiler for allocation to register variables and temporary expression results. If the compiler cannot allocate a register of a required type, spilling occurs. Spilling is the process of moving a register's contents to memory to free the register for another purpose.

Objects of type double, long, long long, or long double are allocated into an odd/even register pair and are always referenced as a register pair (for example, A1:A0). The odd register contains the sign bit, the exponent, and the most significant part of the mantissa. The even register contains the least significant part of the mantissa. The A4 register is used with A5 for passing the first argument if the first argument is a double, long, long long, or long double. The same is true for B4 and B5 for the second parameter, and so on. For more information about argument-passing registers and return registers, see Section 8.4.

Table 8-2. Register Usage

Register	Function Preserved By	Special Uses	Register	Function Preserved By	Special Uses
A0	Parent	–	B0	Parent	–
A1	Parent	–	B1	Parent	–
A2	Parent	–	B2	Parent	–
A3	Parent	Structure register (pointer to returned structure)(1)	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		B16-B31	Parent	
ILC	Child	Loop buffer counter	NRP	Parent	
IRP	Parent		RILC	Child	Loop buffer counter

(1) Structs of size 64 or less are passed by value in registers instead of by reference using a pointer in A3.

All other control registers are not saved or restored by the compiler.

The compiler assumes that control registers not listed in Table 8-2 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.

8.4 Function Structure and Calling Conventions

The C/C++ compiler imposes a strict set of rules on function calls. Except for special run-time support functions, any function that calls or is called by a C/C++ function must follow these rules. Failure to adhere to these rules can disrupt the C/C++ environment and cause a program to fail.

For details on the calling conventions, refer to *The C6000 Embedded Application Binary Interface Application Report* (SPRAB89).

8.4.1 How a Function Makes a Call

A function (parent function) performs the following tasks when it calls another function (child function).

1. Arguments passed to a function are placed in registers or on the stack.

   A function (parent function) performs the following tasks when it calls another function (child function):

   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.

   The C6600 __x128_t type object is aligned to a 64-bit or 128-bit boundary. (See the note in Section 8.6.2.) However, for C6600, if multiple __x128_t arguments are passed, the next __x128_t argument is passed in the first available quad, where the list of available quads has the ordering: A7:A6:A5:A4, B7:B6:B5:B4, A11:A10:A9:A8, B11:B10:B9:B8. If there are no more available quads, the __x128_t goes onto the stack. A subsequent 32-bit, 40-bit, or 64-bit argument can take the first available register or register pair even if an earlier __x128_t argument has been put on the stack.

   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, assuming for C6600 an __x128_t is not passed.) Arguments placed on the stack must be aligned to a value appropriate for their size. An argument that is not declared in a prototype and whose size is less than the size of int is passed as an int. An argument that is a float is passed as double if it has no prototype declared.

   A structure argument is passed as the address of the structure. It is up to the called function to make a local copy.

   For a function declared with an ellipsis indicating that it is called with varying numbers of arguments, the convention is slightly modified. The last explicitly declared argument is passed on the stack, so that its stack address can act as a reference for accessing the undeclared arguments.

   Figure 8-10 shows the register argument conventions.
2. The calling function must save registers A0-A9, B0-B9, A16-A31, and B16-B31 if their values are needed after the call. It should do this by pushing the values onto the stack. See Section 3.2 of the *C6000 Embedded Application Binary Interface Application Report* (SPRAB89A) for details about register conventions.
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.

```
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
__x128_t func5(        __x128_t a);
A7:A6:A5:A4          A7:A6:A5:A4
void func6(int a,    int b,      __x128_t c);
           A4        B4          A11:A10:A9:A8
void func7(int a,    int b,      __x128_t c,    int d,          int e,     int f,   __x128_t g,  int h);
           A4        B4          A11:A10:A9:A8  A6              B6         B8       stack        B10
```

Figure 8-10. Register Argument Conventions

8.4.2 How a Called Function Responds

A called function (child function) must perform the following tasks:

1. The called function (child) allocates enough space on the stack for any local variables, temporary storage areas, and arguments to functions that this function might call. This allocation occurs once at the beginning of the function and may include the allocation of the frame pointer (FP).

   The frame pointer is used to read arguments from the stack and to handle register spilling instructions. If any arguments are placed on the stack or if the frame size exceeds 128K bytes, the frame pointer (A15) is allocated in the following manner:
   a. The old A15 is saved on the stack.
   b. The new frame pointer is set to the current SP (B15).
   c. The frame is allocated by decrementing SP by a constant.
   d. Neither A15 (FP) nor B15 (SP) is decremented anywhere else within this function.

   If the above conditions are not met, the frame pointer (A15) is not allocated. In this situation, the frame is allocated by subtracting a constant from register B15 (SP). Register B15 (SP) is not decremented anywhere else within this function.
2. If the called function calls any other functions, the return address must be saved on the stack. Otherwise, it is left in the return register (B3) and is overwritten by the next function call.
3. If the called function modifies any registers numbered A10 to A15 or B10 to B15, it must save them, either in other registers or on the stack. The called function can modify any other registers without saving them.
4. If the called function expects a structure argument, it receives a pointer to the structure instead. If writes are made to the structure from within the called function, space for a local copy of the structure must be allocated on the stack and the local structure must be copied from the passed pointer to the structure. If no writes are made to the structure, it can be referenced in the called function indirectly through the pointer argument.

You must be careful to declare functions properly that accept structure arguments, both at the point where they are called (so that the structure argument is passed as an address) and at the point where they are declared (so the function knows to copy the structure to a local copy).

5. The called function executes the code for the function.
6. If the called function returns any integer, pointer, or float type, the return value is placed in the A4 register. If the function returns a double, long double, long, or long long type, the value is placed in the A5:A4 register pair. For C6600 if the function returns a __x128_t, the value is placed in A7:A6:A5:A4.

   If the function returns a structure, the caller allocates space for the structure and passes the address of the return space to the called function in A3. To return a structure, the called function copies the structure to the memory block pointed to by the extra argument.

   In this way, the caller can be smart about telling the called function where to return the structure. For example, in the statement s = f(x), where s is a structure and f is a function that returns a structure, the caller can actually make the call as f(&s, x). The function f then copies the return structure directly into s, performing the assignment automatically.

   If the caller does not use the return structure value, an address value of 0 can be passed as the first argument. This directs the called function not to copy the return structure.

   You must be careful to declare functions properly that return structures, both at the point where they are called (so that the extra argument is passed) and at the point where they are declared (so the function knows to copy the result).
7. Any register numbered A10 to A15 or B10 to B15 that was saved in Step 3 is restored.
8. If A15 was used as a frame pointer (FP), the old value of A15 is restored from the stack. The space allocated for the function in Step 1 is reclaimed at the end of the function by adding a constant to register B15 (SP).
9. The function returns by jumping to the value of the return register (B3) or the saved value of the return register.

8.4.3 Accessing Arguments and Local Variables

A function accesses its stack arguments and local nonregister variables indirectly through register A15 (FP) or through register B15 (SP), one of which points to the top of the stack. Since the stack grows toward smaller addresses, the local and argument data for a function are accessed with a positive offset from FP or SP. Local variables, temporary storage, and the area reserved for stack arguments to functions called by this function are accessed with offsets smaller than the constant subtracted from FP or SP at the beginning of the function.

Stack arguments passed to this function are accessed with offsets greater than or equal to the constant subtracted from register FP or SP at the beginning of the function. The compiler attempts to keep register arguments in their original registers if optimization is used or if they are defined with the register keyword. Otherwise, the arguments are copied to the stack to free those registers for further allocation.

For information on whether FP or SP is used to access local variables, temporary storage, and stack arguments, see Section 8.4.2. For more information on the C/C++ System stack, see Section 8.1.2.

8.5 Accessing Linker Symbols in C and C++

See the section on "Linker Symbols" in the *TMS320C6000 Assembly Language Tools User's Guide* for information about referring to linker symbols in C/C++ code.

8.6 Interfacing C and C++ With Assembly Language

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

- Use separate modules of assembled code and link them with compiled C/C++ modules (see Section 8.6.1).
- Use assembly language variables and constants in C/C++ source (see Section 8.6.3).
- Use inline assembly language embedded directly in the C/C++ source (see Section 8.6.5).
- Use intrinsics in C/C++ source to directly call an assembly language statement (see Section 8.6.6).

8.6.1 Using Assembly Language Modules With C/C++ Code

Interfacing C/C++ with assembly language functions is straightforward if you follow the calling conventions defined in Section 8.4, and the register conventions defined in Section 8.3. 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 8.3.
- 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 8.7.
- When you call a C/C++ function from assembly language, load the designated registers with arguments and push the remaining arguments onto the stack as described in Section 8.4.1.

  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. For C6600 __x128_t values are returned in A7:A6: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 write assembly language code, you must use the same linknames as those assigned by the compiler. See Section 7.12 for details.
- 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 8.7.1 for more information.
- The compiler assumes that control registers not listed in Table 8-2 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.

8.6.2 Accessing Assembly Language Functions From C/C++

Functions defined in C++ that will be called from assembly should be defined as extern "C" in the C++ file. Functions defined in assembly that will be called from C++ must be prototyped as extern "C" in C++.

Example 8-1 illustrates a C++ function called main, which calls an assembly language function called asmfunc, Example 8-2. The asmfunc function takes its single argument, adds it to the C++ global variable called gvar, and returns the result.

Example 8-1. Calling an Assembly Language Function From a C/C++ Program

```
extern "C" {
extern int asmfunc(int a); /* declare external asm function */
int gvar = 0;              /* define global variable        */
}
void main()
{
   int I = 5;
   I = asmfunc(I);         /* call function normally       */
```

Example 8-2. Assembly Language Program Called by Example 8-1

```
   .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 8-1, the extern declaration of asmfunc is optional because the return type is int. Like C/C++ functions, you need to declare assembly functions only if they return noninteger values or pass noninteger parameters.

Note

SP Semantics

The stack pointer must always be 8-byte aligned. This is automatically performed by the C compiler and system initialization code in the run-time-support libraries. Any hand-written 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.

Note

Stack Allocation

Even though the compiler guarantees a doubleword alignment of the stack and the stack pointer (SP) points to the next free location in the stack space, there is only enough guaranteed room to store one 32-bit word at that location. The called function must allocate space to store the doubleword.

Note

Alignment of __x128_t Type Data Objects (C6600 Only)

The C6600 provides the 128-bit container type __x128_t. Global data objects of this type are aligned to an 16-byte boundary (128 bits). Local __x128_t variables are allocated on the stack, but are not necessarily aligned on an 16-byte boundary, since their actual alignment depends on the alignment of the stack pointer (SP) and the SP-relative offset of the local __x128_t type object. The compiler aligns the stack to an 8-byte boundary.

8.6.3 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 linker symbol.

8.6.3.1 Accessing Assembly Language Global Variables

Accessing 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 8-4 and Example 8-3 show how you can access a variable defined in .bss.

Example 8-3. 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 8-4. C Program to Access Assembly Language From Example 8-3

```
extern int var1;        /* External variable */
extern far int var2;    /* External variable */
var1 = 1;               /* Use the variable  */
var2 = 1;               /* Use the variable  */
```

8.6.3.2 Accessing Assembly Language Constants

You can define global constants in assembly language by using the .set directive in combination with either the .def or .global directive, 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 **variables** defined in C/C++ or assembly language, the symbol table contains the *address of the value* contained by the variable. When you access an assembly variable by name from C/C++, the compiler gets the value using the address in the symbol table.

For **assembly constants**, however, the symbol table contains the actual *value* of the constant. The compiler cannot tell which items in the symbol table are addresses and which are values. If you access an assembly (or linker) constant by name, the compiler tries to use the value in the symbol table as an address to fetch a value. To prevent this behavior, you must use the & (address of) operator to get the value (_symval). In other words, if x is an assembly language constant, its value in C/C++ is &x. See the section on "Using Linker Symbols in C/C++ Applications" in the *TMS320C6000 Assembly Language Tools User's Guide* for more examples that use _symval.

For more about symbols and the symbol table, refer to the section on "Symbols" in the *TMS320C6000 Assembly Language Tools User's Guide*.

You can use casts and #defines to ease the use of these symbols in your program, as in Example 8-5 and Example 8-6.

Example 8-5. 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 8-6. Assembly Language Program for Example 8-5

```
_table_size  .set10000     ; define the constant
             .global _table_size  ; make it global
```

Because you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example 8-5, int is used. You can reference linker-defined symbols in a similar manner.

8.6.4 Sharing C/C++ Header Files With Assembly Source

You can use the .cdecls assembler directive to share C headers containing declarations and prototypes between C and assembly code. Any legal C/C++ can be used in a .cdecls block and the C/C++ declarations will cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code. For more information, see the C/C++ header files chapter in the *TMS320C6000 Assembly Language Tools User's Guide*.

8.6.5 Using Inline Assembly Language

Within a C/C++ program, you can use the asm statement to insert a single line of assembly language into the assembly language file created by the compiler. A series of asm statements places sequential lines of assembly language into the compiler output with no intervening code. For more information, see Section 7.8.

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.

8.6.6 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);
```

Note

Intrinsic Instructions in C Versus Assembly Language

In some instances, an intrinsic's exact corresponding assembly language instruction may not be used by the compiler. When this is the case, the meaning of the program does not change.

The tables that list intrinsics apply to device families as follows:

Table 8-3. Device Families and Intrinsics Tables

Family	Table 8-5	Table 8-6	Table 8-7
C6400+	Yes		
C6740	Yes	Yes	
C6600	Yes	Yes	Yes

Table 8-4 provides a summary of the C6000 intrinsics clarifying which devices support which intrinsics.

Table 8-4. C6000 C/C++ Intrinsics Support by Device

Intrinsic	C6400+	C6740	C6600
_abs	Yes	Yes	Yes
_abs2	Yes	Yes	Yes
_add2	Yes	Yes	Yes
_add4	Yes	Yes	Yes
_addsub	Yes	Yes	Yes
_addsub2	Yes	Yes	Yes
_amem2	Yes	Yes	Yes
_amem2_const	Yes	Yes	Yes
_amem4	Yes	Yes	Yes
_amem4_const	Yes	Yes	Yes
_amem8	Yes	Yes	Yes
_amem8_const	Yes	Yes	Yes
_amem8_f2	Yes	Yes	Yes
_amem8_f2_const	Yes	Yes	Yes
_amemd8	Yes	Yes	Yes
_amemd8_const	Yes	Yes	Yes
_avg2	Yes	Yes	Yes
_avgu4	Yes	Yes	Yes
_bitc4	Yes	Yes	Yes
_bitr	Yes	Yes	Yes
_ccmatmpy			Yes
_ccmatmpyr1			Yes
_ccmpy32r1			Yes
_clr	Yes	Yes	Yes
_clrr	Yes	Yes	Yes
_cmatmpy			Yes
_cmatmpyr1			Yes
_cmpeq2	Yes	Yes	Yes
_cmpeg4	Yes	Yes	Yes
_cmpgt2	Yes	Yes	Yes
_cmpgtu4	Yes	Yes	Yes
_cmplt2	Yes	Yes	Yes
_cmpltu4	Yes	Yes	Yes
_cmpy	Yes	Yes	Yes
_cmpy32r1			Yes
_cmpyr	Yes	Yes	Yes
_cmpyr1	Yes	Yes	Yes
_cmpysp			Yes
_complex_conjugate_mpysp			Yes
_complex_mpysp			Yes
_crot270			Yes
_crot90			Yes
_dadd			Yes
_dadd2			Yes

Table 8-4. C6000 C/C++ Intrinsics Support by Device (continued)

Intrinsic	C6400+	C6740	C6600
_daddsp			Yes
_dadd_c			Yes
_dapys2			Yes
_davg2			Yes
_davgnr2			Yes
_davgnru4			Yes
_davgu4			Yes
_dccmpyr1			Yes
_dcmpeq2			Yes
_dcmpeq4			Yes
_dcmpgt2			Yes
_dcmpgtu4			Yes
_dccmpy			Yes
_dcmpy			Yes
_dcmpyr1			Yes
_dcrot90			Yes
_dcrot270			Yes
_ddotp4	Yes	Yes	Yes
_ddotp4h			Yes
_ddotph2	Yes	Yes	Yes
_ddotph2r	Yes	Yes	Yes
_ddotpl2	Yes	Yes	Yes
_ddotpl2r	Yes	Yes	Yes
_ddotpsu4h			Yes
_deal	Yes	Yes	Yes
_dinthsp			Yes
_dinthspu			Yes
_dintsp			Yes
_dintspu			Yes
_dmax2			Yes
_dmaxu4			Yes
_dmin2			Yes
_dminu4			Yes
_dmpy2			Yes
_dmpysp			Yes
_dmpysu4			Yes
_dmpyu2			Yes
_dmpyu4			Yes
_dmv	Yes	Yes	Yes
_dmvd			Yes
_dotp2	Yes	Yes	Yes
_dotp4h			Yes
_dotp4hll			Yes
_dotpn2	Yes	Yes	Yes
_dotpnrsu2	Yes	Yes	Yes

Table 8-4. C6000 C/C++ Intrinsics Support by Device (continued)

Intrinsic	C6400+	C6740	C6600
_dotpnrus2	Yes	Yes	Yes
_dotprsu2	Yes	Yes	Yes
_dotpsu4	Yes	Yes	Yes
_dotpus4	Yes	Yes	Yes
_dotpsu4h			Yes
_dotpsu4hll			Yes
_dotpu4	Yes	Yes	Yes
_dpack2	Yes	Yes	Yes
_dpackh2			Yes
_dpackh4			Yes
_dpacklh2			Yes
_dpacklh4			Yes
_dpackl2			Yes
_dpackl4			Yes
_dpackx2	Yes	Yes	Yes
_dpint		Yes	Yes
_dsadd			Yes
_dsadd2			Yes
_dshl			Yes
_dshl2			Yes
_dshr			Yes
_dshr2			Yes
_dshru			Yes
_dshru2			Yes
_dsmpy2			Yes
_dspacku4			Yes
_dspint			Yes
_dspinth			Yes
_dssub			Yes
_dssub2			Yes
_dsub			Yes
_dsub2			Yes
_dsubsp			Yes
_dtol	Yes	Yes	Yes
_dtoll	Yes	Yes	Yes
_dxpnd2			Yes
_dxpnd4			Yes
_ext	Yes	Yes	Yes
_extr	Yes	Yes	Yes
_extu	Yes	Yes	Yes
_extur	Yes	Yes	Yes
_f2tol		Yes	Yes
_f2toll		Yes	Yes
_fabs		Yes	Yes
_fabsf		Yes	Yes

Table 8-4. C6000 C/C++ Intrinsics Support by Device (continued)

Intrinsic	C6400+	C6740	C6600
_fdmvd_f2			Yes
_fdmv_f2	Yes	Yes	Yes
_ftoi	Yes	Yes	Yes
_gmpy	Yes	Yes	Yes
_gmpy4	Yes	Yes	Yes
_hi	Yes	Yes	Yes
_hill	Yes	Yes	Yes
_itod	Yes	Yes	Yes
_itof	Yes	Yes	Yes
_itoll	Yes	Yes	Yes
_labs	Yes	Yes	Yes
_land			Yes
_landn			Yes
_ldotp2	Yes	Yes	Yes
_lmbd	Yes	Yes	Yes
_lnorm	Yes	Yes	Yes
_lo	Yes	Yes	Yes
_loll	Yes	Yes	Yes
_lor			Yes
_lsadd	Yes	Yes	Yes
_lssub	Yes	Yes	Yes
_ltod	Yes	Yes	Yes
_lltod	Yes	Yes	Yes
_lltof2		Yes	Yes
_ltof2		Yes	Yes
_max2	Yes	Yes	Yes
_maxu4	Yes	Yes	Yes
_mfence			Yes
_min2	Yes	Yes	Yes
_minu4	Yes	Yes	Yes
_mem2	Yes	Yes	Yes
_mem2_const	Yes	Yes	Yes
_mem4	Yes	Yes	Yes
_mem4_const	Yes	Yes	Yes
_mem8	Yes	Yes	Yes
_mem8_const	Yes	Yes	Yes
_mem8_f2		Yes	Yes
_mem8_f2_const		Yes	Yes
_memd8	Yes	Yes	Yes
_memd8_const	Yes	Yes	Yes
_mpy	Yes	Yes	Yes
_mpy2ir	Yes	Yes	Yes
_mpy2ll	Yes	Yes	Yes
_mpy32	Yes	Yes	Yes
_mpy32ll	Yes	Yes	Yes

Table 8-4. C6000 C/C++ Intrinsics Support by Device (continued)

Intrinsic	C6400+	C6740	C6600
_mpy32su	Yes	Yes	Yes
_mpy32u	Yes	Yes	Yes
_mpy32us	Yes	Yes	Yes
_mpyh	Yes	Yes	Yes
_mpyhill	Yes	Yes	Yes
_mpyihll	Yes	Yes	Yes
_mpyilll	Yes	Yes	Yes
_mpyhir	Yes	Yes	Yes
_mpyihr	Yes	Yes	Yes
_mpyilr	Yes	Yes	Yes
_mpyhl	Yes	Yes	Yes
_mpyhlu	Yes	Yes	Yes
_mpyhslu	Yes	Yes	Yes
_mpyhsu	Yes	Yes	Yes
_myphu	Yes	Yes	Yes
_mpyhuls	Yes	Yes	Yes
_mpyhus	Yes	Yes	Yes
_mpyidll		Yes	Yes
_mpylh	Yes	Yes	Yes
_mpylhu	Yes	Yes	Yes
_mpylill	Yes	Yes	Yes
_mpylir	Yes	Yes	Yes
_mpylshu	Yes	Yes	Yes
_mpyluhs	Yes	Yes	Yes
_mpysp2dp		Yes	Yes
_mpyspdp		Yes	Yes
_mpysu	Yes	Yes	Yes
_mpysu4ll	Yes	Yes	Yes
_mpyus4ll	Yes	Yes	Yes
_mpyu	Yes	Yes	Yes
_mpyu2			Yes
_mpyu4ll	Yes	Yes	Yes
_mpyus	Yes	Yes	Yes
_mvd	Yes	Yes	Yes
_nassert	Yes	Yes	Yes
_norm	Yes	Yes	Yes
_pack2	Yes	Yes	Yes
_packh2	Yes	Yes	Yes
_packh4	Yes	Yes	Yes
_packhl2	Yes	Yes	Yes
_packl4	Yes	Yes	Yes
_packlh2	Yes	Yes	Yes
_qmpy32			Yes
_qmpysp			Yes
_qsmpy32r1			Yes

Table 8-4. C6000 C/C++ Intrinsics Support by Device (continued)

Intrinsic	C6400+	C6740	C6600
_rcpdp		Yes	Yes
_rcpsp		Yes	Yes
_rsqrdp		Yes	Yes
_rsqrsp		Yes	Yes
_rotl	Yes	Yes	Yes
_rpack2	Yes	Yes	Yes
_sadd	Yes	Yes	Yes
_sadd2	Yes	Yes	Yes
_saddsub	Yes	Yes	Yes
_saddsub2	Yes	Yes	Yes
_saddu4	Yes	Yes	Yes
_saddus2	Yes	Yes	Yes
_saddsu2	Yes	Yes	Yes
_sat	Yes	Yes	Yes
_set	Yes	Yes	Yes
_setr	Yes	Yes	Yes
_shfl	Yes	Yes	Yes
_shfl3	Yes	Yes	Yes
_shl2			Yes
_shlmb	Yes	Yes	Yes
_shr2	Yes	Yes	Yes
_shrmb	Yes	Yes	Yes
_shru2	Yes	Yes	Yes
_smpy	Yes	Yes	Yes
_smpy2ll	Yes	Yes	Yes
_smpy32	Yes	Yes	Yes
_smpyh	Yes	Yes	Yes
_smpyhl	Yes	Yes	Yes
_smpylh	Yes	Yes	Yes
_spack2	Yes	Yes	Yes
_spacku4	Yes	Yes	Yes
_spint		Yes	Yes
_sshl	Yes	Yes	Yes
_sshvl	Yes	Yes	Yes
_sshvr	Yes	Yes	Yes
_ssub	Yes	Yes	Yes
_ssub2	Yes	Yes	Yes
_sub2	Yes	Yes	Yes
_sub4	Yes	Yes	Yes
_subabs4	Yes	Yes	Yes
_subc	Yes	Yes	Yes
_swap2	Yes	Yes	Yes
_swap4	Yes	Yes	Yes
_unpkbu4			Yes
_unpkh2			Yes

Table 8-4. C6000 C/C++ Intrinsics Support by Device (continued)

Intrinsic	C6400+	C6740	C6600
_unpkhu2			Yes
_unpkhu4	Yes	Yes	Yes
_unpklu4	Yes	Yes	Yes
_xorll_c			Yes
_xormpy	Yes	Yes	Yes
_xpnd2	Yes	Yes	Yes
_xpnd4	Yes	Yes	Yes

The intrinsics listed in Table 8-5 can be used on 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.

See Table 8-6 for a list of intrinsics that are specific to C6740 and C6600. See Table 8-7 for a list of C6600-specifiic intrinsics.

Some items listed in the following tables are actually defined in the c6x.h header file as macros that point to intrinsics. This header file is provided in the compiler's "include" directory. Your code must include this header file in order to use the noted macros.

Table 8-5. TMS320C6000 C/C++ Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_abs** (int *src*); __int40_t **_labs** (__int40_t *src*);	**ABS**	Returns the saturated absolute value of src
int **_abs2** (int *src*);	**ABS2**	Calculates the absolute value for each 16-bit value
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.
int **_add4** (int *src1* , int *src2*);	**ADD4**	Performs 2s-complement addition to pairs of packed 8-bit numbers
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.
ushort & **_amem2** (void **ptr*);	**LDHU** **STH**	Allows aligned loads and stores of 2 bytes to memory. The pointer must be aligned to a two-byte boundary.(1)
const ushort & **_amem2_const** (const void **ptr*);	**LDHU**	Allows aligned loads of 2 bytes from memory. The pointer must be aligned to a two-byte boundary.(1)
unsigned & **_amem4** (void **ptr*);	**LDW** **STW**	Allows aligned loads and stores of 4 bytes to memory. The pointer must be aligned to a four-byte boundary.(1)
const unsigned & **_amem4_const** (const void **ptr*);	**LDW**	Allows aligned loads of 4 bytes from memory. The pointer must be aligned to a four-byte boundary.(1)
long long & **_amem8** (void **ptr*);	**LDDW** **STDW**	Allows aligned loads and stores of 8 bytes to memory. The pointer must be aligned to an eight-byte boundary. An LDDW or STDW instruction will be used.
const long long & **_amem8_const** (const void **ptr*);	**LDW/LDW** **LDDW**	Allows aligned loads of 8 bytes from memory. The pointer must be aligned to an eight-byte boundary.(2)
__float2_t & **_amem8_f2**(void * *ptr*);	**LDDW** **STDW**	Allows aligned loads and stores of 8 bytes to memory. The pointer must be aligned to an eight-byte boundary. This is defined as a macro. You must include c6x.h. (2) (1)
const __float2_t & **_amem8_f2_const**(void * *ptr*);	**LDDW**	Allows aligned loads of 8 bytes from memory. The pointer must be aligned to an eight-byte boundary. This is defined as a macro. You must include c6x.h. (2) (1)
double & **_amemd8** (void **ptr*);	**LDDW** **STDW**	Allows aligned loads and stores of 8 bytes to memory. The pointer must be aligned to an eight-byte boundary.(1) (2) An LDDW or STDW instruction will be used.

Table 8-5. TMS320C6000 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
const double & **_amemd8_const** (const void *_ptr_);	**LDW/LDW LDDW**	Allows aligned loads of 8 bytes from memory. The pointer must be aligned to an eight-byte boundary.(1) (2)
int **_avg2** (int _src1_ , int _src2_);	**AVG2**	Calculates the average for each pair of signed 16-bit values
unsigned **_avgu4** (unsigned _src1_ , unsigned _src2_);	**AVGU4**	Calculates the average for each pair of unsigned 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
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.
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 unsigned 8-bit values. Results are packed into the four least-significant bits of the return value.
int **_cmplt2** (int _src1_ , int _src2_);	**CMPLT2**	Swaps operands and calls _cmpgt2. This is defined as a macro. You must include c6x.h.
unsigned **_cmpltu4** (unsigned _src1_ , unsigned _src2_);	**CMPLTU4**	Swaps operands and calls _cmpgtu4. This is defined as a macro. You must include c6x.h.
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.
unsigned **_deal** (unsigned _src_);	**DEAL**	The odd and even bits of src are extracted into two separate 16-bit values.
long long **_dmv** (int _src1_ , int _src2_);	**DMV**	Places src1 in the 32 MSBs of the long long and src2 in the 32 LSBs of the long long. See also _itoll().
int **_dotp2** (int _src1_ , int _src2_); __int40_t **_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. In the case of _dotp2, the signed result is written to a single 32-bit register. In the case of _ldotp2, the signed result is written to a 64-bit register pair.
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 16-bit values of src1 and src2 is subtracted from the product of the upper 16-bit values of src1 and src2. The values in src1 are treated as signed packed quantities; the values in src2 are treated as unsigned packed quantities. 2^15 is added and the result is sign shifted right by 16.
int **_dotpnrus2** (unsigned _src1_ , int _src2_);	**DOTPNRUS2**	Swaps the operands and calls _dotpnrsu2. This is defined as a macro. You must include c6x.h.

Table 8-5. TMS320C6000 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int **_dotprsu2** (int *src1* , unsigned *src2*);	**DOTPRSU2**	The product of the lower 16-bit values of src1 and src2 is added to the product of the upper 16-bit values of src1 and src2. The values in src1 are treated as signed packed quantities; the values in src2 are treated as unsigned packed quantities. 2^15 is added and the result is sign shifted by 16.
int **_dotpsu4** (int *src1* , unsigned *src2*); int **_dotpus4** (unsigned *src1* , int *src2*); unsigned **_dotpu4** (unsigned *src1* , unsigned *src2*);	**DOTPSU4** **DOTPUS4** **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. _dotpus4 is defined as a macro. You must include c6x.h.
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.
__int40_t **_dtol** (double *src*);		Reinterprets double register pair src as an __int40_t (stored as a register pair).
long long **_dtoll** (double *src*);		Reinterprets double register pair src as a long 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.
__float2_t **_fdmv_f2**(float *src1* , float *src2*);	**DMV**	Places src1 in the 32 LSBs of the __float2_t and src2 in the 32 MSBs of the __float2_t. See also _itoll(). This is defined as a macro. You must include c6x.h.
unsigned **_ftoi** (float *src*);		Reinterprets the bits in the float as an unsigned. For example: _ftoi (1.0) == 1065353216U
unsigned **_gmpy** (unsigned *src1* , unsigned *src2*);	**GMPY**	Performs the Galois Field multiply.
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.
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** (__int40_t *src*);		Reinterprets an __int40_t register pair src as a double register pair.
double **_lltod** (long long *src*);		Reinterprets long long register pair src as a double register pair.

Table 8-5. TMS320C6000 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
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** **MAXU4** **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(1)
const ushort & **_mem2_const** (const void * *ptr*);	**LDB/LDB**	Allows unaligned loads of 2 bytes to memory(1)
unsigned & **_mem4** (void * *ptr*);	**LDNW** **STNW**	Allows unaligned loads and stores of 4 bytes to memory(1)
const unsigned & **_mem4_const** (const void * *ptr*);	**LDNW**	Allows unaligned loads of 4 bytes from memory(1)
long long & **_mem8** (void * *ptr*);	**LDNDW** **STNDW**	Allows unaligned loads and stores of 8 bytes to memory(1)
const long long & **_mem8_const** (const void * *ptr*);	**LDNDW**	Allows unaligned loads of 8 bytes from memory(1)
double & **_memd8** (void * *ptr*);	**LDNDW** **STNDW**	Allows unaligned loads and stores of 8 bytes to memory(2) (1)
const double & **_memd8_const** (const void * *ptr*);	**LDNDW**	Allows unaligned loads of 8 bytes from memory(2) (1)
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.
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.
long long **_mpy2ll** (int *src1* , int *src2*);	**MPY2**	Returns the products of the lower and higher 16-bit values in src1 and src2
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.
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.
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.
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.
long long **_mpyihll** (int *src1* , int *src2*); long long **_mpyilll** (int *src1* , int *src2*);	**MPYIH** **MPYIL**	Swaps operands and calls _mpyhill. This is defined as a macro. You must include c6x.h. Swaps operands and calls _mpylill. This is defined as a macro. You must include c6x.h.
int **_mpyihr** (int *src1* , int *src2*); int **_mpyilr** (int *src1* , int *src2*);	**MPYIHR** **MPYILR**	Swaps operands and calls _mpyhir. This is defined as a macro. You must include c6x.h. Swaps operands and calls _mpylir. This is defined as a macro. You must include c6x.h.
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.

Table 8-5. TMS320C6000 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
long long **_mpysu4ll** (int *src1* , unsigned *src2*); long long **_mpyus4ll** (unsigned *src1* , int *src2*); long long **_mpyu4ll** (unsigned *src1* , unsigned *src2*);	**MPYSU4** **MPYUS4** **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. _mpyus4ll is defined as a macro. You must include c6x.h.
int **_mvd** (int *src2*);	**MVD**	Moves the data from src2 to the return value over four cycles using the multiplier pipeline
void **_nassert** (int *src*);		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 *src*); unsigned **_lnorm** (__int40_t *src*);	**NORM**	Returns the number of bits up to the first nonredundant sign bit of src
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 src1 to the left by the amount in src2
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 32-bit output. The 16 MSBs of the shifted src2 is placed in the 16 LSBs of the 32-bit output.
int **_sadd** (int *src1* , int *src2*); __int40_t **_lsadd** (int *src1* , __int40_t *src2*);	**SADD**	Adds src1 to src2 and saturates the result. Returns the result.
int **_sadd2** (int *src1* , int *src2*); int **_saddus2** (unsigned *src1* , int *src2*); int **_saddsu2** (int *src1* , unsigned *src2*);	**SADD2** **SADDUS2** **SADDSU2**	Performs saturated addition between pairs of 16-bit values in src1 and src2. Values for src1 can be signed or unsigned. _saddsu2 is defined as a macro. You must include c6x.h.
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.
unsigned **_saddu4** (unsigned *src1* , unsigned *src2*);	**SADDU4**	Performs saturated addition between pairs of 8-bit unsigned values in src1 and src2.
int **_sat** (__int40_t *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.
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.
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.
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 src1, the quantity is arithmetically or logically shifted right by src2 number of bits. src1 can contain signed or unsigned values.
int **_smpy** (int *src1* , int *src2*); int **_smpyh** (int *src1* , int *src2*); int **_smpyhl** (int *src1* , int *src2*); int **_smpylh** (int *src1* , int *src2*);	**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

Table 8-5. TMS320C6000 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
long long **_smpy2ll** (int *src1* , int *src2*);	**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 **_smpy32** (int *src1* , int *src2*);	**SMPY32**	Returns the 32 MSBs of a 32 by 32 multiply shifted left by 1.
int **_spack2** (int *src1* , int *src2*);	**SPACK2**	Two signed 32-bit values are saturated to 16-bit values and packed into the return value
unsigned **_spacku4** (int *src1* , int *src2*);	**SPACKU4**	Four signed 16-bit values are saturated to 8-bit values and packed into the return value
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 **_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 **_ssub** (int *src1* , int *src2*); __int40_t **_lssub** (int *src1* , __int40_t *src2*);	**SSUB**	Subtracts src2 from src1, saturates the result, and returns the result.
int **_ssub2** (int *src1* , int *src2*);	**SSUB2**	Subtracts the upper and lower halves of src2 from the upper and lower halves of src1 and saturates each result.
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 unsigned 8-bit values
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.
unsigned **_swap4** (unsigned *src*);	**SWAP4**	Exchanges pairs of bytes (an endian swap) within each 16-bit value.
unsigned **_swap2** (unsigned *src*);	**SWAP2**	Calls _packlh2. This is defined as a macro. You must include c6x.h.
unsigned **_unpkhu4** (unsigned *src*);	**UNPKHU4**	Unpacks the two high unsigned 8-bit values into unsigned packed 16-bit values
unsigned **_unpklu4** (unsigned *src*);	**UNPKLU4**	Unpacks the two low unsigned 8-bit values into unsigned packed 16-bit values
unsigned **_xormpy** (unsigned *src1* , unsigned *src2*);	**XORMPY**	Performs a Galois Field multiply
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.

(1) See the *TMS320C6000 Programmer's Guide* for more information.
(2) See Section 8.6.10 for details on manipulating 8-byte data quantities.

The intrinsics listed in Table 8-6 can be used for C6740 and C6600 devices, but not C6400+ devices. The intrinsics listed correspond to the indicated C6000 assembly language instruction(s). See the *TMS320C6000 CPU and Instruction Set Reference Guide* for more information.

See Table 8-5 for a list of generic C6000 intrinsics. See Table 8-7 for a list of C6600-specific intrinsics.

Table 8-6. TMS320C6740 and C6600 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.
__int40_t **_f2tol**(__float2_t *src*);		Reinterprets a __float2_t register pair src as an __int40_t (stored as a register pair). This is defined as a macro. You must include c6x.h.

Table 8-6. TMS320C6740 and C6600 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
__float2_t **_f2toll(**__float2_t *src* **);**		Reinterprets a __float2_t register pair as a long long register pair. This is defined as a macro. You must include c6x.h.
double **_fabs (**double *src* **);** float **_fabsf (**float *src* **);**	**ABSDP** **ABSSP**	Returns absolute value of src.
__float2_t **_lltof2(**long long *src* **);**		Reinterprets a long long register pair as a __float2_t register pair. This is defined as a macro. You must include c6x.h.
__float2_t **_ltof2(**__int40_t *src* **);**		Reinterprets an __int40_t register pair as a __float2_t register pair. This is defined as a macro. You must include c6x.h.
__float2_t & **_mem8_f2(**void * *ptr* **);**	**LDNDW** **STNDW**	Allows unaligned loads and stores of 8 bytes to memory.(1) This is defined as a macro. You must include c6x.h.
const __float2_t & **_mem8_f2_const(**void * *ptr* **);**	**LDNDW** **STNDW**	Allows unaligned loads of 8 bytes from memory.(1) This is defined as a macro. You must include c6x.h.
long long **_mpyidll (**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**	Produces a double-precision floating-point multiply. The result is placed in a register pair.
double**_mpyspdp (**float *src1* **,** double *src2* **);**	**MPYSPDP**	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 *src* **);**	**SPINT**	Converts 32-bit float to 32-bit signed integer, using the rounding mode set by the CSR register.

(1) See Section 8.6.10 for details on manipulating 8-byte data quantities.

The intrinsics listed in Table 8-7 are supported only for C6600 devices. These intrinsics are in addition to those listed in Table 8-5 and Table 8-6. The intrinsics listed correspond to the indicated assembly language instruction(s). See the *TMS320C6000 CPU and Instruction Set Reference Guide* for more information.

Table 8-7. TMS320C6600 C/C++ Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction	Description
	ADDDP	No intrinsic. Use native C: a + b where a and b are doubles.
	ADDSP	No intrinsic. Use native C: a + b where a and b are floats.
	AND	No intrinsic: Use native C: "a & b" where a and b are long longs.
	ANDN	No intrinsic: Use native C: "a & ~b" where a and b are long longs.
	FMPYDP	No intrinsic. Use native C: a * b where a and b are doubles.
	OR	No intrinsic: Use native C: "a | b" where a and b are long longs.
	SUBDP	No intrinsic. Use native C: a - b where a and b are doubles.
	SUBSP	No intrinsic. Use native C: a - b where a and b are floats.
	XOR	No intrinsic: Use native C: "a ^ b" where a and b are long longs. See also _xorll_c().
__x128_t **_ccmatmpy (**long long *src1* **,** __x128_t *src2* **);**	**CCMATMPY**	Multiply the conjugate of 1x2 complex vector by a 2x2 complex matrix, producing two 64-bit results. For details on the __x128_t container type see Section 8.6.7.

Table 8-7. TMS320C6600 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
long long **_ccmatmpyr1 (**long long *src1* , __x128_t *src2* **);**	**CCMATMPYR1**	Multiply the complex conjugate of a 1x2 complex vector by a 2x2 complex matrix, producing two 32-bit complex results.
long long **_ccmpy32r1 (**long long *src1* , long long *src2* **);**	**CCMPY32R1**	32-bit complex conjugate multiply of Q31 numbers with rounding.
__x128_t **_cmatmpy (**long long *src1* , __x128_t *src2* **);**	**CMATMPY**	Multiply a 1x2 vector by a 2x2 complex matrix, producing two 64-bit complex results.
long long **_cmatmpyr1 (**long long *src1* , __x128_t *src2* **);**	**CMATMPYR1**	Multiply a 1x2 complex vector by a 2x2 complex matrix, producing two 32-bit complex results.
long long **_cmpy32r1 (**long long *src1* , long long *src2* **);**	**CMPY32R1**	32-bit complex multiply of Q31 numbers with rounding.
__x128_t **_cmpysp (**__float2_t *src1* , __float2_t *src2* **);**	**CMPYSP**	Perform the multiply operations for a complex multiply of two complex numbers (See also _complex_mpysp and _complex_conjugate_mpysp.)
double **_complex_conjugate_mpysp (**double *src1* , double *src2* **);**	**CMPYSP DSUBSP**	Performs a complex conjugate multiply by performing a CMPYSP and DSUBSP.
double **_complex_mpysp (**double *src1* , double *src2* **);**	**CMPYSP DADDSP**	Performs a complex multiply by performing a CMPYSP and DADDSP.
int **_crot90 (**int *src* **);**	**CROT90**	Rotate complex number by 90 degrees.
int **_crot270 (**int *src* **);**	**CROT270**	Rotate complex number by 270 degrees.
long long **_dadd (**long long *src1* , long long *src2* **);**	**DADD**	Two-way SIMD addition of signed 32-bit values producing two signed 32-bit results.
long long **_dadd2 (**long long *src1* , long long *src2* **);**	**DADD2**	Four-way SIMD addition of packed signed 16-bit values producing four signed 16-bit results. (Two-way _add2)
__float2_t **_daddsp (**__float2_t *src1* , __float2_t *src2* **);**	**DADDSP**	Two-way SIMD addition of 32-bit single precision numbers.
long long **_dadd_c (**scst5 immediate *src1* , long long *src2* **);**	**DADD**	Addition of two signed 32-bit values by a single constant in src2 (-16 to 15) producing two signed 32-bit results.
long long **_dapys2 (**long long *src1* , long long *src2* **);**	**DAPYS2**	Use the sign bit of src1 to determine whether to multiply the four 16-bit values in src2 by 1 or -1. Yields four signed 16-bit results. (If src1 and src2 are the same register pair, it is equivalent to a two-way _abs2).
long long **_davg2 (**long long *src1* , long long *src2* **);**	**DAVG2**	Four-way SIMD average of signed 16-bit values, with rounding. (Two-way _avg2)
long long **_davgnr2 (**long long *src1* , long long *src2* **);**	**DAVGNR2**	Four-way SIMD average of signed 16-bit values, without rounding.
long long **_davgnru4 (**long long *src1* , long long *src2* **);**	**DAVGNRU4**	Eight-way SIMD average of unsigned 8-bit values, without rounding.
long long **_davgu4 (**long long *src1* , long long *src2* **);**	**DAVGU4**	Eight-way SIMD average of unsigned 8-bit values, with rounding. (Two-way _avgu4)
long long **_dccmpyr1 (**long long *src1* , long long *src2* **);**	**DCCMPYR1**	Two-way SIMD complex multiply with rounding (_cmpyr1) with complex conjugate of src2.
unsigned **_dcmpeq2 (**long long *src1* , long long *src2* **);**	**DCMPEQ2**	Four-way SIMD comparison of signed 16-bit values. Results are packed into the four least-significant bits of the return value. (Two-way _cmpeq2)
unsigned **_dcmpeq4 (**long long *src1* , long long *src2* **);**	**DCMPEQ4**	Eight-way SIMD comparison of unsigned 8-bit values. Results are packed into the eight least-significant bits of the return value. (Two-way _cmpeq4)
unsigned **_dcmpgt2 (**long long *src1* , long long *src2* **);**	**DCMPGT2**	Four-way SIMD comparison of signed 16-bit values. Results are packed into the four least-significant bits of the return value. (Two-way _cmpgt2)
unsigned **_dcmpgtu4 (**long long *src1* , long long *src2* **);**	**DCMPGTU4**	Eight-way SIMD comparison of unsigned 8-bit values. Results are packed into the eight least-significant bits of the return value. (Two-way _cmpgtu4)
__x128_t **_dccmpy (**long long *src1* , long long *src2* **);**	**DCCMPY**	Two complex multiply operations on two sets of packed complex numbers, with complex conjugate of src2.

Table 8-7. TMS320C6600 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
__x128_t **_dcmpy (**long long *src1* , long long *src2* **);**	**DCMPY**	Performs two complex multiply operations on two sets of packed complex numbers. (Two-way SIMD _cmpy)
long long **_dcmpyr1 (**long long *src1* , long long *src2* **);**	**DCMPYR1**	Two-way SIMD complex multiply with rounding (_cmpyr1).
long long **_dcrot90 (**long long *src* **);**	**DCROT90**	Two-way SIMD version of _crot90.
long long **_dcrot270 (**long long *src* **);**	**DCROT270**	Two-way SIMD version of _crot270.
long long **_ddotp4h (**__x128_t *src1* , __x128_t *src2* **);**	**DDOTP4H**	Performs two dot-products between four sets of packed 16-bit values. (Two-way _dotp4h)
long long **_ddotpsu4h (**__x128_t *src1* , __x128_t *src2* **);**	**DDOTPSU4H**	Performs two dot-products between four sets of packed 16-bit values. (Two-way _dotpsu4h)
__float2_t **_dinthsp (**int *src* **);**	**DINTHSP**	Converts two packed signed 16-bit values into two single-precision floating point values.
__float2_t **_dinthspu (**unsigned *src* **);**	**DINTHSPU**	Converts two packed unsigned 16-bit values into two single-precision float point values.
__float2_t **_dintsp(**long long *src* **);**	**DINTSP**	Converts two 32-bit signed integers to two single-precision float point values.
__float2_t **_dintspu(**long long *src* **);**	**DINTSPU**	Converts two 32-bit unsigned integers to two single-precision float point values.
long long **_dmax2 (**long long *src1* , long long *src2* **);**	**DMAX2**	Four-way SIMD maximum of 16-bit signed values producing four signed 16-bit results. (Two-way _max2)
long long **_dmaxu4 (**long long *src1* , long long *src2* **);**	**DMAXU4**	8-way SIMD maximum of unsigned 8-bit values producing eight unsigned 8-bit results. (Two-way _maxu4)
long long **_dmin2 (**long long *src1* , long long *src2* **);**	**DMIN2**	Four-way SIMD minimum of signed 16-bit values producing four signed 16-bit results. (Two-way _min2)
long long **_dminu4 (**long long *src1* , long long *src2* **);**	**DMINU4**	8-way SIMD minimum of unsigned 8-bit values producing eight unsigned 8-bit results. (Two-way _minu4)
__x128_t **_dmpy2 (**long long *src1* , long long *src2* **);**	**DMPY2**	Four-way SIMD multiply of signed 16-bit values producing four signed 32-bit results. (Two-way _mpy2)
__float2_t **_dmpysp (**__float2_t *src1* , __float2_t *src2* **);**	**DMPYSP**	Two-way single precision floating point multiply producing two single-precision results.
__x128_t **_dmpysu4 (**long long *src1* , long long *src2* **);**	**DMPYSU4**	Eight-way SIMD multiply of signed 8-bit values by unsigned 8-bit values producing eight signed 16-bit results. (Two-way _mpysu4)
__x128_t **_dmpyu2 (**long long *src1* , long long *src2* **);**	**DMPYU2**	Four-way SIMD multiply of unsigned 16-bit values producing four unsigned 32-bit results. (Two-way _mpyu2)
__x128_t **_dmpyu4 (**long long *src1* , long long *src2* **);**	**DMPYU4**	Eight-way SIMD multiply of signed 8-bit values producing eight signed 16-bit results. (Two-way _mpyu4)
long long **_dmvd (**int *src1* , int *src2* **);**	**DMVD**	Places src1 in the low register of the long long and src2 in the high register of the long long. Takes four cycles. See also _dmv(), _fdmv_f2, and _itoll().
int **_dotp4h (**long long *src1* , long long *src2* **);**	**DOTP4H**	Multiply two sets of four signed 16-bit values and return the 32-bit sum.
long long **_dotp4hll (**long long *src1* , long long *src2* **);**	**DOTP4H**	Multiply two sets of four signed 16-bit values and return the 64-bit sum.
int **_dotpsu4h (**long long *src1* , long long *src2* **);**	**DOTPSU4H**	Multiply four signed 16-bit values by four unsigned 16-bit values and return the 32-bit sum.
long long **_dotpsu4hll (**long long *src1* , long long *src2* **);**	**DOTPSU4H**	Multiply four signed 16-bit values by four unsigned 16-bit values and return the 64-bit sum.
long long **_dpackh2 (**long long *src1* , long long *src2* **);**	**DPACKH2**	Two-way _packh2.
long long **_dpackh4 (**long long *src1* , long long *src2* **);**	**DPACKH4**	Two-way _packh4.
long long **_dpacklh2 (**long long *src1* , long long *src2* **);**	**DPACKLH2**	Two-way _packlh2.

Table 8-7. TMS320C6600 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
long long **_dpacklh4 (**unsigned *src1* , unsigned *src2* **);**	**DPACKLH4**	Performs a _packl4 and a _packh4. The output of the _packl4 is in the low register of the result and the output of the _packh4 is in the high register of the result.
long long **_dpackl2 (**long long *src1* , long long *src2* **);**	**DPACKL2**	Two-way _packl2.
long long **_dpackl4 (**long long *src1* , long long *src2* **);**	**DPACKL4**	Two-way _packl4.
long long **_dsadd (**long long *src1* , long long *src2* **);**	**DSADD**	Two-way SIMD saturated addition of signed 32-bit values producing two signed 32-bit results. (Two-way _sadd)
long long **_dsadd2 (**long long *src1* , long long *src2* **);**	**DSADD2**	Four-way SIMD saturated addition of signed 16-bit values producing four signed 16-bit results. (Two-way _sadd2)
long long **_dshl (**long long *src1* , unsigned *src2* **);**	**DSHL**	Shift-left of two signed 32-bit values by a single value in the src2 argument.
long long **_dshl2 (**long long *src1* , unsigned *src2* **);**	**DSHL2**	Shift-left of four signed 16-bit values by a single value in the src2 argument. (Two-way _shl2)
long long **_dshr (**long long *src1* , unsigned *src2* **);**	**DSHR**	Shift-right of two signed 32-bit values by a single value in the src2 argument.
long long **_dshr2 (**long long *src1* , unsigned *src2* **);**	**DSHR2**	Shift-right of four signed 16-bit values by a single value in the src2 argument. (Two-way _shr2)
long long **_dshru (**long long *src1* , unsigned *src2* **);**	**DSHRU**	Shift-right of two unsigned 32-bit values by a single value in the src2 argument.
long long **_dshru2 (**long long *src1* , unsigned *src2* **);**	**DSHRU2**	Shift-right of four unsigned 16-bit values by a single value in the src2 argument. (Two-way _shru2)
__x128_t **_dsmpy2 (**long long *src1* , long long *src2* **);**	**DSMPY2**	Four-way SIMD multiply of signed 16-bit values with 1-bit left-shift and saturate producing four signed 32-bit results. (Two-way _smpy2)
long long **_dspacku4 (**long long *src1* , long long *src2* **);**	**DSPACKU4**	Two-way _spacku4.
long long **_dspint (**__float2_t *src* **);**	**DSPINT**	Converts two packed single-precision floating point values to two signed 32-bit values.
unsigned **_dspinth (**__float2_t *src* **);**	**DSPINTH**	Converts two packed single-precision floating point values to two packed signed 16-bit values.
long long **_dssub (**long long *src1* , long long *src2* **);**	**DSSUB**	Two-way SIMD saturated subtraction of 32-bit signed values producing two signed 32-bit results.
long long **_dssub2 (**long long *src1* , long long *src2* **);**	**DSSUB2**	Four-way SIMD saturated subtraction of signed 16-bit values producing four signed 16-bit results. (Two-way _ssub2)
long long **_dsub (**long long *src1* , long long *src2* **);**	**DSUB**	Two-way SIMD subtraction of 32-bit signed values producing two signed 32-bit results.
long long **_dsub2 (**long long *src1* , long long *src2* **);**	**DSUB2**	Four-way SIMD subtraction of signed 16-bit values producing four signed 16-bit results. (Two-way _sub2)
__float2_t **_dsubsp (**__float2_t *src1* , __float2_t *src2* **);**	**DSUBSP**	Two-way SIMD subtraction of 32-bit single precision numbers.
long long **_dxpnd2 (**unsigned *src* **);**	**DXPND2**	Expand four lower bits to four 16-bit fields.
long long **_dxpnd4 (**unsigned *src* **);**	**DXPND4**	Expand eight lower bits to eight 8-bit fields.
__float2_t **_fdmvd_f2(**float *src1* , float *src2* **);**	**DMVD**	Places src1 in the low register of the __float2_t and src2 in the high register of the __float2_t. Takes four cycles. See also _dmv(), _dmvd(), and _itoll(). This is defined as a macro. You must include c6x.h.
int **_land (**int *src1* , int *src2* **);**	**LAND**	Logical AND of src1 and src2.
int **_landn (**int *src1* , int *src2* **);**	**LANDN**	Logical AND of src1 and NOT of src2; i.e. src1 AND ~src2.
int **_lor (**int *src1* , int *src2* **);**	**LOR**	Logical OR of src1 and src2.
void **_mfence();**	**MFENCE**	Stall CPU while memory system is busy.
long long **_mpyu2 (**unsigned *src1* , unsigned *src2* **);**	**MPYU2**	Two-way SIMD multiply of unsigned 16-bit values producing two unsigned 32-bit results.

Table 8-7. TMS320C6600 C/C++ Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
__x128_t **_qmpy32 (**__x128_t *src1* **,** __x128_t *src2* **);**	**QMPY32**	Four-way SIMD multiply of signed 32-bit values producing four 32-bit results. (Four-way _mpy32)
__x128_t **_qmpysp (**__x128_t *src1* **,** __x128_t *src2* **);**	**QMPYSP**	Four-way SIMD 32-bit single precision multiply producing four 32-bit single precision results.
__x128_t **_qsmpy32r1 (**__x128_t *src1* **,** __x128_t *src2* **);**	**QSMPY32R1**	4-way SIMD fractional 32-bit by 32-bit multiply where each result value is shifted right by 31 bits and rounded. This normalizes the result to lie within -1 and 1 in a Q31 fractional number system.
unsigned **_shl2 (**unsigned *src1* **,** unsigned *src2* **);**	**SHL2**	Shift-left of two signed 16-bit values by a single value in the src2 argument.
long long **_unpkbu4 (**unsigned *src* **);**	**UNPKBU4**	Unpack four unsigned 8-bit values into four unsigned 16-bit values. (See also _unpklu4 and _unpkhu4)
long long **_unpkh2 (**unsigned *src* **);**	**UNPKH2**	Unpack two signed 16-bit values to two signed 32-bit values.
long long **_unpkhu2 (**unsigned *src* **);**	**UNPKHU2**	Unpack two unsigned 16-bit values to two unsigned 32-bit values.
long long **_xorll_c (**scst5 immediate *src1* **,** long long *src2* **);**	**XOR**	XOR src1 with the upper and lower 32-bit portions of src2 (SIMD XOR by constant).

8.6.7 The __x128_t Container Type

The __x128_t container type is available when compiling for C6600 only, It stores 128-bits of data and its use is necessary when performing certain SIMD operations on C6600. Also, note the leading double underscore. When using the __x128_t container type, you must include c6x.h.

This type can be used to define objects that can be used with certain C6600 intrinsics. (See Table 8-7.) The object can be filled and manipulated using various intrinsics. The type is not a full-fledged built-in type (like long long), and so various native C operations are not allowed. Think of this type as a struct with private members and special manipulation functions.

When the compiler puts a __x128_t object in the register file, the __x128_t object takes four registers (a register *quad*). Objects of type __x128_t are aligned to a 64-bit boundary in memory.

When an __x128_t object is passed on the stack, it is placed on a 64-bit boundary relative to the beginning of the stack. (The stack itself is aligned to a 64-bit boundary by default.) See the note in Section 8.6.2 for details.

The following operations are supported:

- Declare a __x128_t global object (for example: `__x128_t a;`). By default, it will be put in the .far section.
- Declare a __x128_t local object (for example: `__x128_t a;`). It will be put on the stack.
- Declare a __x128_t global/local pointer (for example: `__x128_t *a;`).
- Declare an array of __x128_t objects (for example: `__x128_t a[10];`).
- Declare a __x128_t type as a member of a struct, class, or union.
- Assign a __x128_t object to another __x128_t object.
- Pass a __x128_t object to a function (including variadic argument functions). (Pass by value.)
- Return a __x128_t object from a function.
- Use 128-bit manipulation intrinsics to set and extract contents (see Table 8-8).

The following operations are not supported:

- Native-type operations on __x128_t objects, such as +, -, *, etc.
- Cast an object to a __x128_t type.
- Access the elements of a __x128_t using array or struct notation.
- Pass a __x128_t object to I/O functions like printf. Instead, extract the values from the __x128_t object by using appropriate intrinsics.

Example 8-7. The __x128_t Container Type

```
#include <c6x.h>
#include <stdio.h>
__x128_t mpy_four_way_example(__x128_t s, int a, int b, int c, int d)
{
  __x128_t t = _ito128(a, b, c, d);       // Pack values into a __x128_t
  __x128_t results = _qmpy32(s, t);       // Perform a four-way SIMD multiply

  int lowest32  = _get32_128(results, 0); // Extract lowest reg of __x128_t
  int highest32 = _get32_128(results, 3); // Extract highest reg of __x128_t
  printf("lowest  = %d\n", lowest32);
  printf("highest = %d\n", highest32);

  return results;
}
```

Note

Include c6x.h With Type __x128_t or __float2_t

When using the __x128_t container type, or __float2_t typedef, or any intrinsics involving __float2_t, you must include c6x.h.

Table 8-8. Vector-in-Scalar Support C/C++ Compiler v7.2 Intrinsics

C/C++ Compiler Intrinsic	Description
Creation	
__x128_t **_ito128 (**unsigned *src1* **,** unsigned *src2* **,** unsigned *src3* **,** unsigned *src4* **);**	Creates __x128_t from (u)int (reg+3, reg+2, reg+1, reg+0)
__x128_t **_fto128 (**float *src1* **,** float *src2* **,** float *src3* **,** float *src4* **);**	Creates __x128_t from float (reg+3, reg+2, reg+1, reg+0)
__x128_t **_llto128 (**long long *src1* **,** long long *src2* **);**	Creates __x128_t from two long longs
__x128_t **_dto128 (**double *src1* **,** double *src2* **);**	Creates __x128_t from two doubles
__x128_t **_f2to128(**__float2_t *src1* **,** __float2_t *src2* **);**	Creates __x128_t from two __float2_t objects. This is defined as a macro. You must include c6x.h.
__x128_t **_dup32_128 (**int *src* **);**	Creates __x128_t from duplicating src1
__float2_t **_ftof2(**float *src1* **,** float *src2* **);**	Creates __float2_t from two floats. This is defined as a macro. You must include c6x.h.
Extraction	
float **_hif (**double *src* **);**	Extracts upper float from double
float **_lof (**double *src* **);**	Extracts lower float from double
float **_hif2(**__float2_t *src* **);**	Extracts upper float from __float2_t. This is defined as a macro. You must include c6x.h.
float **_lof2(**__float2_t *src* **);**	Extracts lower float from __float2_t. This is defined as a macro. You must include c6x.h.
long long **_hi128 (**__x128_t *src* **);**	Extracts upper two registers of register quad
double **_hid128 (**__x128_t *src* **);**	Extracts upper two registers of register quad
__float2_t **_hif2_128(**__x128_t *src* **);**	Extracts upper two registers of register quad. This is defined as a macro. You must include c6x.h.
long long **_lo128 (**__x128_t *src* **);**	Extracts lower two registers of register quad
double **_lod128 (**__x128_t *src* **);**	Extracts lower two registers of register quad
__float2_t **_lof2_128(**__x128_t *src* **);**	Extracts lower two registers of register quad. This is defined as a macro. You must include c6x.h.
unsigned **_get32_128 (**__x128_t *src* **, 0);**	Extracts first register of register quad (base reg + 0)
unsigned **_get32_128 (**__x128_t *src* **, 1);**	Extracts second register of register quad (base reg + 1)
unsigned **_get32_128 (**__x128_t *src* **, 2);**	Extracts third register of register quad (base reg + 2)
unsigned **_get32_128 (**__x128_t *src* **, 3);**	Extracts fourth register of register quad (base reg + 3)
float **_get32f_128 (**__x128_t *src* **, 0);**	Extracts first register of register quad (base reg + 0)
float **_get32f_128 (**__x128_t *src* **, 1);**	Extracts second register of register quad (base reg + 1)
float **_get32f_128 (**__x128_t *src* **, 2);**	Extracts third register of register quad (base reg + 2)
float **_get32f_128 (**__x128_t *src* **, 3);**	Extracts fourth register of register quad (base reg + 3)

8.6.8 The __float2_t Container Type

The __float2_t container type should be used (instead of double) to store two floats. There are manipulation intrinsics to create and manipulate objects with the __float2_t type (see Table 8-8). The run-time-support file, c6x.h, must be included when using __float2_t or when using any of the __float2_t manipulation intrinsics.

Recommendations for using the __float2_t type:

- Use __float2_t to store two floats. Do not use double.
- Use long long to store 64-bit packed integer data. Do not use double or __float2_t for packed integer data.

8.6.9 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_interrupts (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.

Note

Overwrites CSR

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.

The _restore_interrupts( ) intrinsic does not use the RINT instruction.

8.6.10 Using Unaligned Data and 64-Bit Values

The C6000 has support for unaligned loads and stores of 64-bit and 32-bit values via the use of the _mem8, _memd8, and _mem4 intrinsics. The _lo and _hi intrinsics are useful for extracting the two 32-bit portions from a 64-bit double. The _loll and _hill intrinsics are useful for extracting the two 32-bit portions from a 64-bit long long.

For 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 get the upper 32 bits of a __float2_t, use _hif2().
- To get the lower 32 bits of a __float2_t, use _lof2().
- To create a long long value, use the _itoll(int high32bits, int low32bits) intrinsic.
- To create a __float2_t value, use the _ftof2(float high32bits, float low32bits) intrinsic.

Example 8-8 shows the usage of the _mem8 intrinsic.

Example 8-8. 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;
}
```

8.6.11 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 examples generate the same assembly code:

- **Example 1**

```
int dot_product(short *x, short *y, short z)
{
    int *w_x = (int *)x;
    int *w_y = (int *)y;
    int sum1 = 0, sum2 = 0, I;
    for (I = 0; I < z/2; I++)
    {
        sum1 += _mpy(w_x[i], w_y[i]);
        sum2 += _mpyh(w_x[i], w_y[i]);
    }
    return (sum1 + sum2);
}
```

- **Example 2**

```
int dot_product(short *x, short *y, short z)
{
    int sum = 0, I;
    _nassert (((int)(x) & 0x3) == 0);
    _nassert (((int)(y) & 0x3) == 0);
    #pragma MUST_ITERATE(20, , 4);
    for (I = 0; I < z; I++) sum += x[i] * y[i];
    return sum;
}
```

Note

C++ Syntax for _nassert

In C++ code, _nassert is part of the standard namespace. Thus, the correct syntax is std::_nassert().

8.6.12 Methods to Align Data

In the following code, the _nassert tells the compiler, for every invocation of f(), that ptr is aligned to an 8-byte boundary. Such an assertion often leads to the compiler producing code which operates on multiple data values with a single instruction, also known as SIMD (single instruction multiple data) optimization.

```
void f(short *ptr)
{
   _nassert((int) ptr % 8 == 0)
   ; a loop operating on data accessed by ptr
}
```

The following subsections describe methods you can use to ensure the data referenced by ptr is aligned. You have to employ one of these methods at every place in your code where f() is called.

8.6.12.1 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);
```

Such an array is automatically aligned to 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(_TMS320C6X)
    #define ALIGNED_ARRAY(ptr) _nassert((int) ptr % 8 == 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 for all C6000 devices or if you port the code to another target.

8.6.12.2 Offset from the Base of an Array

A more rare case is to pass the address of an offset from an array, for example:

```
f(&buffer[3]);
```

This code passes an unaligned address to ptr, thus violating the presumption coded in the _nassert(). There is no direct remedy for this case. Avoid this practice whenever possible.

8.6.12.3 Dynamic Memory Allocation

Ordinary dynamic memory allocation guarantees that the allocated memory is properly aligned for any scalar object of a native type (for instance, it is correctly aligned for a long double or long long int), but does not guarantee any larger alignment. For example:

```
buffer = calloc(100, sizeof(short))
```

To get a stricter alignment, use the function memalign with the desired alignment. To get an alignment of 256 bytes for example:

```
buffer = memalign(256, 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), **256);**

See the *TMS320C6000 DSP/BIOS Help* for more information about BIOS memory allocation routines and the *segid* parameter in particular.

8.6.12.4 Member of a Structure or Class

Arrays which are members of a structure or a class are aligned only as the base type of the array requires. The automatic alignment described in Section 8.6.12.1 does not occur.

Example 8-9. An Array in a Structure

```
   struct s
   {
      ...
      short buf1[50];
      ...
   }  g;
...
f(g.buf1);
```

Example 8-10. An Array in a Class

```
    class c
    {
       public :
          short buf1[50];
          void mfunc(void);
       ...
    };
    void c::mfunc()
    {
       f(buf1);
       ...
    }
```

To align an array in a structure, place it inside a union with a dummy object that has the desired alignment. If you want 8 byte alignment, use a "long long" dummy field. For example:

```
struct s
{
    union u
    {   long long dummy;  /* 8-byte alignment */
        short buffer[50]; /* also 8-byte alignment */
    } u;
    ...
};
```

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
{
    union u
    {   long long dummy;  /* 8-byte alignment      */
        short buffer[50]; /* also 8-byte alignment */
        short buf2[50];   /* 4-byte alignment      */
        ...
    } u;
};
```

Because the size of buf1 is 50 * 2-bytes per short = 100 bytes, and 100 is an even multiple of 4, not 8, buf2 is only aligned on a 4-byte boundary. Padding buf1 out to 52 elements makes buf2 8-byte aligned.

Within a structure or class, there is no way to enforce an array alignment greater than 8. For the purposes of SIMD optimization, this is not necessary.

Note

Alignment With Program-Level Optimization

In most cases program-level optimization (see Section 4.4) entails compiling all of your source files with a single invocation of the compiler, while using the -pm -o3 options. This allows the compiler to see all of your source code at once, thus enabling optimizations that are rarely applied otherwise. Among these optimizations is seeing that, for instance, all of the calls to the function f() are passing the base address of an array to ptr, and thus ptr is always correctly aligned for SIMD optimization. In such a case, the _nassert() is not required. The compiler automatically determines that ptr must be aligned, and produces the optimized SIMD instructions.

8.6.13 SAT Bit Side Effects

The saturated intrinsic operations define the SAT bit if saturation occurs. The SAT bit can be set and cleared from C/C++ code by accessing the control status register (CSR). The compiler uses the following steps for generating code that accesses the SAT bit:

1. The SAT bit becomes undefined by a function call or a function return. This means that the SAT bit in the CSR is valid and can be read in C/C++ code until a function call or until a function returns.
2. If the code in a function accesses the CSR, then the compiler assumes that the SAT bit is live across the function, which means:
   - The SAT bit is maintained by the code that disables interrupts around software pipelined loops.
   - Saturated instructions cannot be speculatively executed.
3. If an interrupt service routine modifies the SAT bit, then the routine should be written to save and restore the CSR.

8.6.14 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 RESTORE_AMR macros should be used in all interrupts (see Section 8.7.3).
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.

8.6.15 Floating Point and Saturation Control Register Side Effects

When performing floating point operations on a floating-point architecture or when performing saturating operations, status bits in certain control registers may be set. In particular, status bits may be set in the FADCR, FAUCR, FMCR, CSR, and SSR registers, hereafter referred to as the "status bit control registers". These bits can be set and cleared from C/C++ code by writing to or reading from these registers, as shown in Example 7-1.

The compiler may *speculate* instructions under certain circumstances. Speculation is an optimization technique in which the compiler causes an instruction to execute earlier than may be normally expected in order to improve performance. When the compiler detects that an instruction that may write to a status bit control register is used in the function but the associated control register is not read in the enclosing function (including inlining), the compiler assumes it is free to speculate the instruction.

If the program uses instructions that may write to the status bit control registers and then reads the associated control register later in the function (and not in the function in which the status bit control register is set), you can use the --assume_control_regs_read option to prevent the compiler from speculating instructions that may set status bits in status bit control registers.

If an interrupt service routine modifies (or may modify) any bit in a status bit control register, the interrupt service routine should be written to save and restore that floating point control register.

8.7 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.

8.7.1 Saving the SGIE Bit

The compiler may use the 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.

8.7.2 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.The compiler will save and restore the ILC and RILC control registers if needed.

8.7.3 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 7.9.20 and Section 7.5.4, respectively.

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.

```
#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 your 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.

The compiler saves and restores the ILC and RILC control registers if needed.

For floating point architectures, you are responsible for handling the floating point control registers FADCR, FAUCR and FMCR. If you are reading bits out of the floating pointer control registers, and if the interrupt service routine (or any called function) performs floating point operations, then the relevant floating point control registers should be saved and restored. No macros are provided for these registers, as simple assignment to and from an unsigned int temporary will suffice.

8.7.4 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 8-2 are saved, because the C/C++ function can modify them.

8.8 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 provided in the `lib/src` source directory. The source code has comments that describe the operation of the functions. You can extract, inspect, and modify any of the math functions. Be sure, however, that you follow the calling conventions and register-saving rules outlined in this chapter. Table 8-9 summarizes the run-time-support functions used for arithmetic.

Table 8-9. C6000 Run-Time-Support Arithmetic Functions

Type	Function	Description
float	__c6xabi_cvtdf (double)	Convert double to float
int	__c6xabi_fixdi (double)	Convert double to signed integer
long	__c6xabi_fixdi (double)	Convert double to long
long long	__c6xabi_fixdlli (double)	Convert double to long long
uint	__c6xabi_fixdlli (double)	Convert double to unsigned integer
ulong	__c6xabi_fixdul (double)	Convert double to unsigned long
ulong long	__c6xabi_fixdull (double)	Convert double to unsigned long long
double	__c6xabi_cvtfd (float)	Convert float to double
int	__c6xabi_fixfi (float)	Convert float to signed integer
long	__c6xabi_fixfli (float)	Convert float to long
long long	__c6xabi_fixflli (float)	Convert float to long long
uint	__c6xabi_fixfu (float)	Convert float to unsigned integer
ulong	__c6xabi_fixful (float)	Convert float to unsigned long
ulong long	__c6xabi_fixfull (float)	Convert float to unsigned long long
double	__c6xabi_fltid (int)	Convert signed integer to double
float	__c6xabi_fltif (int)	Convert signed integer to float
double	__c6xabi_fltud (uint)	Convert unsigned integer to double
float	__c6xabi_fltuf (uint)	Convert unsigned integer to float
double	__c6xabi_fltlid (long)	Convert signed long to double
float	__c6xabi_fltlif (long)	Convert signed long to float
double	__c6xabi_fltuld (ulong)	Convert unsigned long to double
float	__c6xabi_fltulf (ulong)	Convert unsigned long to float
double	__c6xabi_fltllid (long long)	Convert signed long long to double
float	__c6xabi_fltllif (long long)	Convert signed long long to float
double	__c6xabi_fltulld (ulong long)	Convert unsigned long long to double
float	__c6xabi_fltullf (ulong long)	Convert unsigned long long to float
double	__c6xabi_absd (double)	Double absolute value
float	__c6xabi_absf (float)	Float absolute value
long	__c6xabi_labs (long)	Long absolute value
long long	__c6xabi_llabs (long long)	Long long absolute value
double	__c6xabi_negd (double)	Double negate value
float	__c6xabi_negf (float)	Float negate value
long long	__c6xabi_negll (long)	Long long negate value
long long	__c6xabi_llshl (long long)	Long long shift left
long long	__c6xabi_llshr (long long)	Long long shift right

Table 8-9. C6000 Run-Time-Support Arithmetic Functions (continued)

Type	Function	Description
ulong long	__c6xabi_llshru (ulong long)	Unsigned long long shift right
double	__c6xabi_addd (double, double)	Double addition
double	__c6xabi_cmpd (double, double)	Double comparison
double	__c6xabi_divd (double, double)	Double division
double	__c6xabi_mpyd (double, double)	Double multiplication
double	__c6xabi_subd (double, double)	Double subtraction
float	__c6xabi_addf (float, float)	Float addition
float	__c6xabi_cmpf (float, float)	Float comparison
float	__c6xabi_divf (float, float)	Float division
float	__c6xabi_mpyf (float, float)	Float multiplication
float	__c6xabi_subf (float, float)	Float subtraction
int	__c6xabi_divi (int, int)	Signed integer division
int	__c6xabi_remi (int, int)	Signed integer remainder
uint	__c6xabi_divu (uint, uint)	Unsigned integer division
uint	__c6xabi_remu (uint, uint)	Unsigned integer remainder
long	__c6xabi_divli (long, long)	Signed long division
long	__c6xabi_remli (long, long)	Signed long remainder
ulong	__c6xabi_divul (ulong, ulong)	Unsigned long division
ulong	__c6xabi_remul (ulong, ulong)	Unsigned long remainder
long long	__c6xabi_divlli (long long, long long)	Signed long long division
long long	__c6xabi_remlli (long long, long long)	Signed long long remainder
ulong long	__c6xabi_mpyll (long long, long long)	Unsigned long long multiplication
ulong long	__c6xabi_divull (ulong long, ulong long)	Unsigned long long division
ulong long	__c6xabi_remull (ulong long, ulong long)	Unsigned long long remainder

8.9 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 files. 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 linker input files.

When C/C++ programs are linked, the linker sets the entry point value in the executable output file to the symbol c_int00. This does not, however, set the hardware to automatically vector to c_int00 at reset (see the *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide*, the *TMS320C6740 CPU and Instruction Set Reference Guide*, or the *TMS320C66x+ DSP 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. Performs C autoinitialization of global/static variables. For more information, see Section 8.9.2.
3. 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).
4. Calls C++ initialization routines for file scope construction from the global constructor table. For more information, see Section 8.9.2.6.
5. Calls the main() function 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.

8.9.1 Boot Hook Functions for System Pre-Initialization

Boot hooks are points at which you may insert application functions into the C/C++ boot process. Default boot hook functions are provided with the run-time support (RTS) library. However, you can implement customized versions of these boot hook functions, which override the default boot hook functions in the RTS library if they are linked before the run-time library. Such functions can perform any application-specific initialization before continuing with the C/C++ environment setup.

Note that the TI-RTOS operating system uses custom versions of the boot hook functions for system setup, so you should be careful about overriding these functions if you are using TI-RTOS.

The following boot hook functions are available:

_system_pre_init(): This function provides a place to perform application-specific initialization. It is invoked after the stack pointer is initialized but before any C/C++ environment setup is performed. By default, _system_pre_init() should return a non-zero value. The default C/C++ environment setup is bypassed if _system_pre_init() returns 0.

_system_post_cinit(): This function is invoked during C/C++ environment setup, after C/C++ global data is initialized but before any C++ constructors are called. This function should not return a value.

8.9.2 Automatic Initialization of Variables

Any global variables declared as preinitialized 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. Internally, the compiler and linker coordinate to produce compressed initialization tables. Your code should not access the initialization table.

8.9.2.1 Zero Initializing Variables

In ANSI C, global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ compiler supports preinitialization of uninitialized variables by default. This can be turned off by specifying the linker option --zero_init=off.

Zero initialization takes place only if the --rom_model linker option, which causes autoinitialization to occur, is used. If you use the --ram_model option for linking, the linker does not generate initialization records, and the loader must handle both data and zero initialization.

8.9.2.2 Direct Initialization

The compiler uses direct initialization to initialize global variables. For example, consider the following C code:

```
int i    = 23;
int a[5] = { 1, 2, 3, 4, 5 };
```

The compiler allocates the variables 'i' and 'a[] to .data section and the initial values are placed directly.

```
        .global i
        .data
        .align  4
i:
        .field          23,32                   ; i @ 0
        .global a
        .data
        .align  4
a:
        .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
```

Each compiled module that defines static or global variables contains these .data sections. The linker treats the .data section like any other initialized section and creates an output section. In the load-time initialization model, the sections are loaded into memory and used by the program. See Section 8.9.2.5.

In the run-time initialization model, the linker uses the data in these sections to create initialization data and an additional compressed initialization table. The boot routine processes the initialization table to copy data from load addresses to run addresses. See Section 8.9.2.3.

8.9.2.3 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 linker creates a compressed initialization table and initialization data from the direct initialized sections in the compiled module. The table and data are used by the C/C++ boot routine to initialize variables in RAM using the table and data in ROM.

Figure 8-11 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

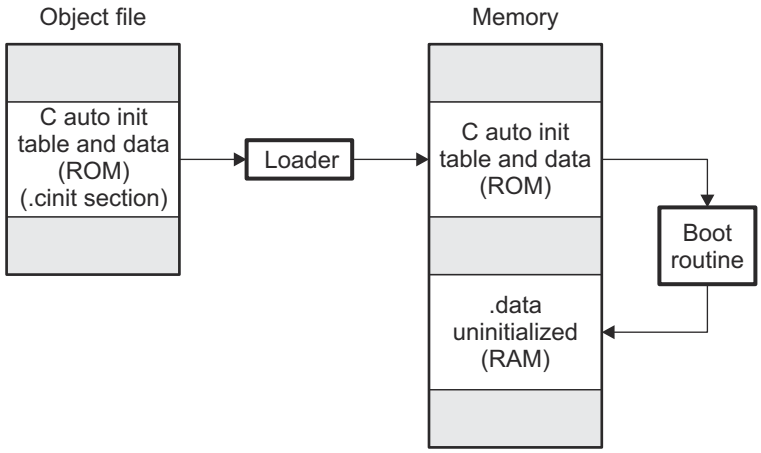


Figure 8-11. Autoinitialization at Run Time

8.9.2.4 Autoinitialization Tables

The compiled object files do not have initialization tables. The variables are initialized directly. The linker, when the --rom_model option is specified, creates C auto initialization table and the initialization data. The linker creates both the table and the initialization data in an output section named .cinit.

The autoinitialization table has the following format:

_TI_CINIT_Base:

32-bit load address	32-bit run address
⋮	⋮
32-bit load address	32-bit run address

_TI_CINIT_Limit:

The linker defined symbols __TI_CINIT_Base and __TI_CINIT_Limit point to the start and end of the table, respectively. Each entry in this table corresponds to one output section that needs to be initialized. The initialization data for each output section could be encoded using different encoding.

The load address in the C auto initialization record points to initialization data with the following format:

8-bit index	Encoded data

The first 8-bits of the initialization data is the handler index. It indexes into a handler table to get the address of a handler function that knows how to decode the following data.

The handler table is a list of 32-bit function pointers.

_TI_Handler_Table_Base:

32-bit handler 1 address
⋮
32-bit handler n address

_TI_Handler_Table_Limit:

The *encoded data* that follows the 8-bit index can be in one of the following format types. For clarity the 8-bit index is also depicted for each format.

8.9.2.4.1 Length Followed by Data Format

8-bit index	24-bit padding	32-bit length (N)	N byte initialization data (not compressed)

The compiler uses 24-bit padding to align the length field to a 32-bit boundary. The 32-bit length field encodes the length of the initialization data in bytes (N). N byte initialization data is not compressed and is copied to the run address as is.

The run-time support library has a function __TI_zero_init() to process this type of initialization data. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

8.9.2.4.2 Zero Initialization Format

8-bit index	24-bit padding	32-bit length (N)

The compiler uses 24-bit padding to align the length field to a 32-bit boundary. The 32-bit length field encodes the number of bytes to be zero initialized.

The run-time support library has a function __TI_zero_init() to process the zero initialization. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

8.9.2.4.3 Run Length Encoded (RLE) Format

8-bit index	Initialization data compressed using run length encoding

The data following the 8-bit index is compressed using Run Length Encoded (RLE) format. uses a simple run length encoding that can be decompressed using the following algorithm:

1. Read the first byte, Delimiter (D).
2. Read the next byte (B).
3. If B != D, copy B to the output buffer and go to step 2.
4. Read the next byte (L).
   a. If L == 0, then length is either a 16-bit, a 24-bit value, or we've reached the end of the data, read next byte (L).
      i. If L == 0, length is a 24-bit value or the end of the data is reached, read next byte (L).
         1. If L == 0, the end of the data is reached, go to step 7.
         2. Else L <<= 16, read next two bytes into lower 16 bits of L to complete 24-bit value for L.
      ii. Else L <<= 8, read next byte into lower 8 bits of L to complete 16-bit value for L.
   b. Else if L > 0 and L < 4, copy D to the output buffer L times. Go to step 2.
   c. Else, length is 8-bit value (L).
5. Read the next byte (C); C is the repeat character.
6. Write C to the output buffer L times; go to step 2.
7. End of processing.

The run-time support library has a routine __TI_decompress_rle24() to decompress data compressed using RLE. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

Note

RLE Decompression Routine

The previous decompression routine, __TI_decompress_rle(), is included in the run-time-support library for decompressing RLE encodings generated by older versions of the linker.

8.9.2.4.4 Lempel-Ziv-Storer-Szymanski Compression (LZSS) Format

8-bit index	Initialization data compressed using LZSS

The data following the 8-bit index is compressed using LZSS compression. The run-time support library has the routine __TI_decompress_lzss() to decompress the data compressed using LZSS. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

8.9.2.4.5 Sample C Code to Process the C Autoinitialization Table

The run-time support boot routine has code to process the C autoinitialization table. The following C code illustrates how the autoinitialization table can be processed on the target.

```
typedef void (*handler_fptr)(const unsigned char *in,
unsigned char *out);
#define HANDLER_TABLE   __TI_Handler_Table_Base
extern  unsigned int   HANDLER_TABLE;
extern  unsigned char *__TI_CINIT_Base;
extern  unsigned char *__TI_CINIT_Limit;
void auto_initialize()
{
   unsigned char **table_ptr;
   unsigned char **table_limit;
   /*-------------------------------------------------------------*/
   /* Check if Handler table has entries.                         */
   /*-------------------------------------------------------------*/
   if (&__TI_Handler_Table_Base >= &__TI_Handler_Table_Limit)
      return;
   /*-------------------------------------------------------------*/
   /* Get the Start and End of the CINIT Table.                   */
   /*-------------------------------------------------------------*/
   table_ptr   = (unsigned char **)&__TI_CINIT_Base;
   table_limit = (unsigned char **)&__TI_CINIT_Limit;
   while (table_ptr < table_limit)
   {
      /*----------------------------------------------------------*/
      /* 1. Get the Load and Run address.                         */
      /* 2. Read the 8-bit index from the load address.           */
      /* 3. Get the handler function pointer using the index from */
      /*    handler table.                                        */
      /*----------------------------------------------------------*/
      unsigned char *load_addr   = *table_ptr++;
      unsigned char *run_addr    = *table_ptr++;
      unsigned char  handler_idx = *load_addr++;
      handler_fptr   handler     =
                   (handler_fptr)(&HANDLER_TABLE)[handler_idx];
      /*----------------------------------------------------------*/
      /* 4. Call the handler and pass the pointer to the load data*/
      /*    after index and the run address.                      */
      /*----------------------------------------------------------*/
      (*handler)((const unsigned char *)load_addr, run_addr);
   }
}
```

8.9.2.5 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 does not generate C autoinitialization tables and data. The direct initialized sections (.data) in the compiled object files are combined according to the linker command file to generate initialized output sections. The loader loads the initialized output sections into memory. After the load, the variables are assigned their initial values.

Since the linker does not generate the C autoinitialization tables, no boot time initialization is performed.

Figure 8-12 illustrates the initialization of variables at load time.

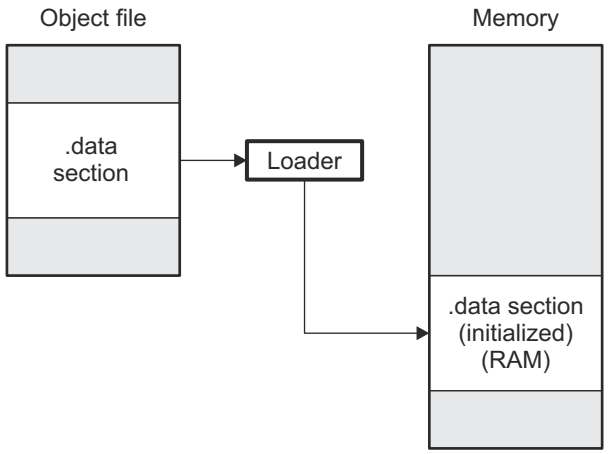


Figure 8-12. Initialization at Load Time

8.9.2.6 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 .init_array. The linker combines the .init_array section form each input file to form a single table in the .init_array section. The boot routine uses this table to execute the constructors. The linker defines two symbols to identify the combined .init_array table as shown below. This table is not null terminated by the linker.

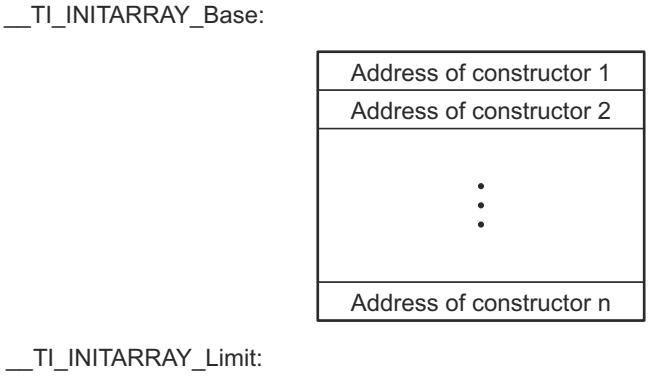


Figure 8-13. Constructor Table

8.10 Support for Multi-Threaded Applications

The compiler supports various features for multi-threaded applications. These features assume an underlying runtime operating system that provides thread management services.

8.10.1 Compiling with OpenMP

The compiler implements support for the OpenMP 3.0 API and portions of the OpenMP 4.0 specification. To enable support for OpenMP use the --openmp or --omp compiler option.

OpenMP is the industry standard for shared memory parallel programming. It provides portable high-level programming constructs that enable users to easily expose a program's task and loop level parallelism in an incremental fashion. With OpenMP, users specify the parallelization strategy for a program at a high level by annotating the program code with compiler directives that specify how a region of code is executed by a team of threads. The compiler works out the detailed mapping of the computation to the machine. The OpenMP programming API enables the programmer to perform the following:

- Create and manage threads
- Assign and distribute work (tasks) to threads

- Specify which data is shared among threads and which data is private
- Coordinate thread access to shared data

OpenMP is a thread-based programming language. The master thread executes the sequential parts of a program. When the master thread encounters a parallel region, it forks a team of worker threads that along with the master thread execute in parallel.

The OpenMP API is made up of directives (#pragmas), function calls, and environment variables. The compiler translates the OpenMP API into multi-threaded code with calls to a custom runtime library that implements support for thread management, shared memory and synchronization. For further details on the OpenMP API (including the API specification), please refer to http://www.openmp.org. For details about TI's support for OpenMP, including descriptions of supported pragmas, see the TI OpenMP Accelerator Model online documentation.

The OpenMP runtime for SYS/BIOS (OMP) library implements the OpenMP solution stack. Currently, OpenMP is supported on TI DSPs only for SYS/BIOS operating system. All OpenMP programs must be linked with the OMP run-time library found in the BIOS-MCSDK 2.1.

8.10.2 Multi-Threading Runtime Support

Compiling with --openmp causes a thread-safe RTS library to be automatically selected by the linker. If you have a non-OpenMP multi-threaded application, you can cause the application to link with a thread-safe version of the RTS library by using the --multithread option instead of the --openmp option.

8.10.2.1 Runtime Thread Safety

Thread safety involves creation, initialization, maintenance, and destruction of thread-private data. It also requires that accesses to data that is shared between threads must be protected. That is, only one thread should be allowed to access a piece of shared data at a given time. An additional issue that needs to be addressed on multi-core devices that have a private data cache (L1D cache on C6600 devices, for example) is that copies of shared data that exist among private data caches and shared memory must be kept coherent. This means that if a thread reads a piece of shared data into the private data cache on a processor, it must invalidate any local copies of that data that exist in the local data cache before accessing or modifying the data. This ensures that the currently executing thread will only access the latest available copy of the shared data.

8.10.2.2 Thread Creation, Initialization, and Termination

Thread libraries are responsible for allocating a thread-local area of memory when a thread is created, then initializing any thread-private data objects that reside in that thread-local area, and when the thread is terminated, the thread library must free the thread-local area that was allocated for thread-private data.

8.10.2.3 Thread Local Storage (TLS)

The compiler supports the __thread qualifier to identify a data object that is to be given thread-private storage. To access a variable that has been identified as thread-local, the compiler will rely on support of the runtime operating system's thread library to find the run-time location of a thread-local variable. Specifically, the thread library must provide an implementation of the __c6xabi_get_tp() function. The runtime operating system's thread library will provide the address of a TLS block that has been allocated on behalf of the currently executed thread and then the compiler can access data in the TLS block with knowledge of where a given thread-local variable exists within the TLS block.

Further information about TLS data objects can be found in the *C6000 Embedded Application Binary Interface Application Report* (SPRAB89A).

8.10.2.4 Accessing Shared Data

Accesses to shared data objects must be protected within a critical region which prevents a second thread from entering the critical region of code while the first thread is accessing a shared data object. We also need to be concerned about data coherency between copies of shared data that may exist in both shared memory and in private data caches as mentioned above in Section 8.10.2.1.

Chapter 9

Using Run-Time-Support Functions and Building Libraries

Some of the features of C/C++ (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are provided as an ANSI/ISO C/C++ standard library, rather than as part of the compiler itself. The TI implementation of this library is the run-time-support library (RTS). The C/C++ compiler implements the ISO standard library except for those facilities that handle signal and 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 9.1 and Section 9.2.

A library-build utility is provided with the code generation tools that lets you create customized run-time-support libraries. This process is described in Section 9.4.

9.1 C and C++ Run-Time Support Libraries

TMS320C6000 compiler releases include pre-built run-time support (RTS) libraries that provide all the standard capabilities. Separate libraries are provided for each target CPU version, big and little endian support, and C++ exception support. See Section 9.1.8 for information on the library-naming conventions.

The run-time-support library contains the following:

- ANSI/ISO C/C++ standard library
- C I/O library
- Low-level support functions that provide I/O to the host operating system
- Fundamental arithmetic routines
- System startup routine, _c_int00
- Compiler helper functions (to support language features that are not directly efficiently expressible in C/C++)

The run-time-support libraries do not contain functions involving signals and locale issues.

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 7.1.

TI does not provide documentation that covers the functionality of the C++ library. TI suggests 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

9.1.1 Linking Code With the Object Library

When you link your program, you must specify the object library as one of the linker input files so that references to the I/O and run-time-support functions can be resolved. You can either specify the library or allow the compiler to select one for you. See Section 6.3.1 for further information.

When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see the*TMS320C6000 Assembly Language Tools User's Guide*.

C, C++, and mixed C and C++ programs can use the same run-time-support library. Run-time-support functions and variables that can be called and referenced from both C and C++ will have the same linkage.

9.1.2 Header Files

You must use the header files provided with the compiler run-time support when using functions from C/C++ standard library. Set the C6X_C_DIR environment variable to the include directory where the tools are installed.

The following header files provide TI extensions to the C standard:

- c6x.h -- Provides intrinsic definitions.
- c6x_vec.h -- Provides support for OpenCL-style vector data types and built-in functions used by the TI compiler.
- cpy_tbl.h -- Declares the copy_in() RTS function, which is used to move code or data from a load location to a separate run location at run-time. This function helps manage overlays.
- _data_synch.h -- Declares functions used by the RTS library to help with shared data synchronization. For example, these functions are used when flushing the local data cache to global shared memory.
- file.h -- Declares functions used by low-level I/O functions in the RTS library.
- gsm.h -- Provides basic DSP operations and GSM math operations defined by the European Telecommunications Standards Institute (ETSI).
- _lock.h -- Used when declaring system-wide mutex locks. This header file is deprecated; use _reg_mutex_api.h and _mutex.h instead.

- memory.h -- Provides the memalign() function, which is not required by the C standard.
- _mutex.h -- Declares functions used by the RTS library to help facilitate mutexes for specific resources that are owned by the RTS. For example, these functions are used for heap or file table allocation.
- _pthread.h -- Declares low-level mutex infrastructure functions and provides support for recursive mutexes.
- _reg_mutex_api.h -- Declares a function that can be used by an RTOS to register an underlying lock mechanism and/or thread ID mechanism that is implemented in the RTOS but is called indirectly by the RTS' _mutex.h functions.
- _reg_synch_api.h -- Declares a function that can be used by an RTOS to register an underlying cache synchronization mechanism that is implemented in the RTOS but is called indirectly by the RTS' _data_synch.h functions.
- strings.h -- Provides additional string functions, including bcmp(), bcopy(), bzero(), ffs(), index(), rindex(), strcasecmp(), and strncasecmp(). See the v7.6 release notes for details on these functions.
- _tls.h -- Provides some useful macros for declaring objects that need to be in thread-local storage if the __thread qualifier is supported. If the __thread qualifier is not enabled, these become normal declarations.
- vect.h -- Provides macros to support 128-bit vectors.

The following standard C header files are provided with the compiler: assert.h, complex.h, ctype.h, errno.h, float.h, inttypes.h, iso646.h, limits.h, locale.h, math.h, setjmp.h, signal.h, stdarg.h, stdbool.h, stddef.h, stdint.h, stdio.h, stdlib.h, string.h, time.h, wchar.h, and wctype.h.

The following standard C++ header files are provided with the compiler: algorithm, bitset, cassert, cctype, cerrno, cfloat, ciso646, climits, clocale, cmath, complex, csetjmp, csignal, cstdarg, cstddef, cstdio, cstdlib, cstring, ctime, cwchar, cwctype, deque, exception, fstream, functional, hash_map, hash_set, iomanip, ios, iosfwd, iostream, istream, iterator, limits, list, locale, map, memory, new, numeric, ostream, queue, rope, set, sstream, stack, stdexcept, streambuf, string, strstream, typeinfo, utility, valarray, and vector.

The following header files are for use with older C++ code: fstream.h, iomanip.h, iostream.h, new.h, stdiostream.h, stl.h, and strstream.h.

The following header files are for internal use by TI components and should not be directly included by your applications: _data_synch.h, _fmt_specifier.h, _isfuncdcl.h, _isfuncdef.h, _mutex.h, _pthread.h, _tls.h, access.h, c60asm.i, cpp_inline_math.h, elf_linkage.h, elfnames.h, linkage.h, mathf.h, mathl.h, pprof.h, unaccess.h, wchar.hx, xcomplex, xdebug, xhash, xiosbase, xlocale, xlocinfo, xlocinfo.h, xlocmes, xlocmon, xlocnum, xloctime, xmemory, xstddef, xstring, xtree, xutility, xwcc.h, ymath.h, and yvals.h,

9.1.3 Modifying a Library Function

You can inspect or modify library functions by examining the source code in the lib/src subdirectory of the compiler installation. For example, `C:\ti\ccsv7\tools\compiler\c6000_#.#.#\lib\src`.

One you have located the relevant source code, change the specific function file and rebuild the library.

You can use this source tree to rebuild the rts64plus.lib library or to build a new library. See Section 9.1.8 for details on library naming and Section 9.4 for details on building

9.1.4 Support for String Handling

The RTS library provides the standard C header file <string.h> as well as the POSIX header file <strings.h>, which provides additional functions not required by the C standard. The POSIX header file <strings.h> provides:

- bcmp(), which is equivalent to memcmp()
- bcopy(), which is equivalent to memmove()
- bzero(), which is equivalent to memset(.., 0, ...);
- ffs(), which finds the first bit set and returns the index of that bit
- index(), which is equivalent to strchr()
- rindex(), which is equivalent to strrchr()
- strcasecmp() and strncasecmp(), which perform case-insensitive string comparisons

In addition, the header file <string.h> provides one additional function not required by the C standard.

- strdup(), which duplicates a string by dynamically allocating memory (as if by using malloc) and copying the string to this allocated memory

9.1.5 Minimal Support for Internationalization

The library includes the header files <locale.h>, <wchar.h>, and <wctype.h>, which provide APIs to support non-ASCII character sets and conventions. Our implementation of these APIs is limited in the following ways:

- The library has minimal support for wide and multibyte characters. The type wchar_t is implemented as unsigned short. 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. See Section 7.4 for more information about extended character sets.
- 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.

9.1.6 Support for Time and Clock Functions

The compiler RTS library supports two low-level time-related standard C functions in `time.h`:

- `clock_t clock(void);`
- `time_t time(time_t *timer);`

The time() function returns the wall-clock time. The clock() function returns the number of clock cycles since the program began executing; it has nothing to do with wall-clock time.

The default implementations of these functions require that the program be run under CCS or a similar tool that supports the CIO System Call Protocol. If CIO is not available and you need to use one of these functions, you must provide your own definition of the function.

The clock() function returns the number of clock cycles since the program began executing. This information might be available in a register on the device itself, but the location varies from platform to platform. The compiler's RTS library provides an implementation that uses the CIO System Call Protocol to communicate with CCS, which figures out how to compute the right value for this device.

If CCS is not available, you must provide an implementation of the clock() function that gathers clock cycle information from the appropriate location on the device.

The time() function returns the real-world time, in terms of seconds since an epoch.

On many embedded systems, there is no internal real-world clock, so a program needs to discover the time from an external source. The compiler's RTS library provides an implementation that uses the CIO System Call Protocol to communicate with CCS, which provides the real-world time.

If CCS is not available, you must provide an implementation of the time() function that finds the time from some other source. If the program is running under an operating system, that operating system should provide an implementation of time().

The time() function returns the number of seconds since an *epoch*. On POSIX systems, the epoch is defined as the number of seconds since midnight UTC January 1, 1970. However, the C standard does not require any particular epoch, and the default TI version of time() uses a different epoch: midnight UTC-6 (CST) Jan 1, 1900. Also, the default TI time_t type is a 32-bit type, while POSIX systems typically use a 64-bit time_t type.

The RTS library provides a non-default implementation of the time() function that uses the midnight UTC January 1, 1970 epoch and the 64-bit time_t type, which is then a typedef for __type64_t.

If your code works with raw time values, you can handle the epoch issue in one of the following ways:

- Use the default time() function with the 1900 epoch and 32-bit time_t type. A separate __time64_t type is available in this case.
- Define the macro __TI_TIME_USES_64. The time() function will use the 1970 epoch and the 64-bit time_t type, in which case time_t is a typedef for __type64_t.

Table 9-1. Differences between __time32_t and __time64_t

	__time32_t	__time64_t
Epoch (start)	Jan. 1, 1900 CST-0600	Jan. 1, 1970 UTC-0000
End date	Feb. 7, 2036 06:28:14	year 292277026596
Sign	Unsigned, so cannot represent dates before the epoch.	Signed, so can represent dates before the epoch.

9.1.7 Allowable Number of Open Files

In the <stdio.h> header file, the value for the macro FOPEN_MAX has the value of the macro _NFILE, which is set to 10. The impact is that you can only have 10 files simultaneously open at one time (including the pre-defined streams - stdin, stdout, stderr).

The C standard requires that the minimum value for the FOPEN_MAX macro is 8. The macro determines the maximum number of files that can be opened at one time. The macro is defined in the stdio.h header file and can be modified by changing the value of the _NFILE macro and recompiling the library.

9.1.8 Library Naming Conventions

By default, the linker uses automatic library selection to select the correct run-time-support library (see Section 6.3.1.1) for your application. If you select the library manually, you must select the matching library using a naming scheme like the following:

rts*trg*[*endian*][*abi*][eh].lib

trg	The device family of the C6000 architecture that the library was built for. This can be one of the following: 64plus, 6740, or 6600.	
endian	Indicates endianness:	
	(blank)	Little-endian library
	e	Big-endian library
abi	Indicates the application binary interface (ABI) used. Although the COFF file format is no longer supported, the library filename still contains "_elf" to distinguish the EABI libraries from older COFF libraries.	
	_elf	EABI
eh	Indicates whether the library has exception handling support	
	(blank)	exception handling not supported
	_eh	exception handling support

9.2 The C I/O Functions

The C I/O functions make it possible to access the host's operating system to perform I/O. The capability to perform I/O on the host gives you more options when debugging and testing code.

The I/O functions are logically divided into layers: high level, low level, and device-driver level.

With properly written device drivers, the C-standard high-level I/O functions can be used to perform I/O on custom user-defined devices. This provides an easy way to use the sophisticated buffering of the high-level I/O functions on an arbitrary device.

Note

Debugger Required for Default HOST: For the default HOST device to work, there must be a debugger to handle the C I/O requests; the default HOST device cannot work by itself in an embedded system. To work in an embedded system, you will need to provide an appropriate driver for your system.

Note

C I/O Mysteriously Fails: If there is not enough space on the heap for a C I/O buffer, operations on the file will silently fail. If a call to printf() mysteriously fails, this may be the reason. The heap needs to be at least large enough to allocate a block of size BUFSIZ (defined in stdio.h) for every file on which I/O is performed, including stdout, stdin, and stderr, plus allocations performed by the user's code, plus allocation bookkeeping overhead. Alternately, declare a char array of size BUFSIZ and pass it to setvbuf to avoid dynamic allocation. To set the heap size, use the --heap_size option when linking (refer to the *Linker Description* chapter in the *TMS320C6000 Assembly Language Tools User's Guide*).

Note

Open Mysteriously Fails: The run-time support limits the total number of open files to a small number relative to general-purpose processors. If you attempt to open more files than the maximum, you may find that the open will mysteriously fail. You can increase the number of open files by extracting the source code from rts.src and editing the constants controlling the size of some of the C I/O data structures. The macro _NFILE controls how many FILE (fopen) objects can be open at one time (stdin, stdout, and stderr count against this total). (See also FOPEN_MAX.) The macro _NSTREAM controls how many low-level file descriptors can be open at one time (the low-level files underlying stdin, stdout, and stderr count against this total). The macro _NDEVICE controls how many device drivers are installed at one time (the HOST device counts against this total).

9.2.1 High-Level I/O Functions

The high-level functions are the standard C library of stream I/O routines (printf, scanf, fopen, getchar, and so on). These functions call one or more low-level I/O functions to carry out the high-level I/O request. The high-level I/O routines operate on FILE pointers, also called *streams*.

Portable applications should use only the high-level I/O functions.

To use the high-level 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>
void 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 main.out from the run-time-support library:

```
cl6x main.c -z --heap_size=1000 --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.

9.2.1.1 Formatting and the Format Conversion Buffer

The internal routine behind the C I/O functions—such as printf(), vsnprintf(), and snprintf()—reserves stack space for a format conversion buffer. The buffer size is set by the macro FORMAT_CONVERSION_BUFFER, which is defined in format.h. Consider the following issues before reducing the size of this buffer:

- The default buffer size is 510 bytes. If MINIMAL is defined, the size is set to 32, which allows integer values without width specifiers to be printed.
- Each conversion specified with %xxxx (except %s) must fit in FORMAT_CONVERSION_BUFSIZE. This means any individual formatted float or integer value, accounting for width and precision specifiers, needs to fit in the buffer. Since the actual value of any representable number should easily fit, the main concern is ensuring the width and/or precision size meets the constraints.
- The length of converted strings using %s are unaffected by any change in FORMAT_CONVERSION_BUFSIZE. For example, you can specify `printf("%s value is %d", some_really_long_string, intval)` without a problem.
- The constraint is for each individual item being converted. For example a format string of `%d item1 %f item2 %e item3` does not need to fit in the buffer. Instead, each converted item specified with a % format must fit.
- There is no buffer overrun check.

9.2.2 Overview of Low-Level I/O Implementation

The low-level functions are comprised of seven 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 are designed to be appropriate for all I/O methods, even those which are not actually disk files. Abstractly, all I/O channels can be treated as files, although some operations (such as lseek) may not be appropriate. See Section 9.2.3 for more details.

The low-level functions are inspired by, but not identical to, the POSIX functions of the same names.

The low-level functions operate on file descriptors. A file descriptor is an integer returned by open, representing an opened file. Multiple file descriptors may be associated with a file; each has its own independent file position indicator.

open

Open File for I/O

Syntax

#include <file.h>

int open (const char * *path* **, unsigned** *flags* **, int** *file_descriptor* **);**

Description

The open function opens the file specified by *path* and prepares it for I/O.

- The *path* is the filename of the file to be opened, including an optional directory path and an optional device specifier (see Section 9.2.5).
- The *flags* are attributes that specify how the 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 */
```

  Low-level I/O routines allow or disallow some operations depending on the flags used when the file was opened. Some flags may not be meaningful for some devices, depending on how the device implements files.
- The *file_descriptor* is assigned by open to an opened file.

  The next available file descriptor is assigned to each new file opened.

Return Value

The function returns one of the following values:

non-negative file descriptor	if successful
-1	on failure

close

Close File for I/O

Syntax

#include <file.h>

int close (int *file_descriptor* **);**

Description

The close function closes the file associated with *file_descriptor*.

The *file_descriptor* is the number assigned by open to an opened file.

Return Value

The return value is one of the following:

0	if successful
-1	on failure

read

Read Characters from a File

Syntax

#include <file.h>

int read (int *file_descriptor* **, char *** *buffer* **, unsigned** *count* **);**

Description

The read function reads *count* characters into the *buffer* from the file associated with *file_descriptor*.

- The *file_descriptor* is the number assigned by open to an opened file.
- The *buffer* is where the read characters are placed.
- The *count* is the number of characters to read from the file.

Return Value

The function returns one of the following values:

0	if EOF was encountered before any characters were read
#	number of characters read (may be less than *count*)
-1	on failure

write

Write Characters to a File

Syntax

#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 file associated with *file_descriptor*.

- The *file_descriptor* is the number assigned by open to an opened file.
- The *buffer* is where the characters to be written are located.
- The *count* is the number of characters to write to the file.

Return Value

The function returns one of the following values:

#	number of characters written if successful (may be less than *count*)
-1	on failure

lseek

Set File Position Indicator

Syntax for C

#include <file.h>

off_t lseek (int *file_descriptor* **, off_t** *offset* **, int** *origin* **);**

Description

The lseek function sets the file position indicator for the given file to a location relative to the specified origin. The file position indicator measures the position in characters from the beginning of the file.

- The *file_descriptor* is the number assigned by open to an opened file.
- 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 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 value is one of the following:

#	new value of the file position indicator if successful
(off_t)-1	on failure

unlink

Delete File

Syntax

#include <file.h>

int unlink (const char * *path* **);**

Description

The unlink function deletes the file specified by *path*. Depending on the device, a deleted file may still remain until all file descriptors which have been opened for that file have been closed. See Section 9.2.3.

The *path* is the filename of the file, including path information and optional device prefix. (See Section 9.2.5.)

Return Value

The function returns one of the following values:

0	if successful
-1	on failure

rename

Rename File

Syntax for C

#include {<stdio.h> | <file.h>}

int rename (const char * *old_name* **, const char *** *new_name* **);**

Syntax for C++

#include {<cstdio> | <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.

Note

The optional device specified in the new name must match the device of the old name. If they do not match, a file copy would be required to perform the rename, and rename is not capable of this action.

Return Value

The function returns one of the following values:

0	if successful
-1	on failure

Note

Although rename is a low-level function, it is defined by the C standard and can be used by portable applications.

9.2.3 Device-Driver Level I/O Functions

At the next level are the device-level drivers. They map directly to the low-level I/O functions. The default device driver is the HOST device driver, which uses the debugger to perform file operations. The HOST device driver is automatically used for the default C streams stdin, stdout, and stderr.

The HOST device driver shares a special protocol with the debugger running on a host system so that the host can perform the C I/O requested by the program. Instructions for C I/O operations that the program wants to perform are encoded in a special buffer named _CIOBUF_ in the .cio section. The debugger halts the program at a special breakpoint (C$$IO$$), reads and decodes the target memory, and performs the requested operation. The result is encoded into _CIOBUF_, the program is resumed, and the target decodes the result.

The HOST device is implemented with seven functions, HOSTopen, HOSTclose, HOSTread, HOSTwrite, HOSTlseek, HOSTunlink, and HOSTrename, which perform the encoding. Each function is called from the low-level I/O function with a similar name.

A device driver is composed of seven required functions. Not all function need to be meaningful for all devices, but all seven must be defined. Here we show the names of all seven functions as starting with DEV, but you may choose any name except for HOST.

DEV_open

Open File for I/O

Syntax

int DEV_open (const char * *path* **, unsigned** *flags* **, int** *llv_fd* **);**

Description

This function finds a file matching *path* and opens it for I/O as requested by *flags*.

- The *path* is the filename of the file to be opened. If the name of a file passed to open has a device prefix, the device prefix will be stripped by open, so DEV_open will not see it. (See Section 9.2.5 for details on the device prefix.)
- The *flags* are attributes that specify how the 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 */
```

  See POSIX for further explanation of the flags.
- The *llv_fd* is treated as a suggested low-level file descriptor. This is a historical artifact; newly-defined device drivers should ignore this argument. This differs from the low-level I/O open function.

This function must arrange for information to be saved for each file descriptor, typically including a file position indicator and any significant flags. For the HOST version, all the bookkeeping is handled by the debugger running on the host machine. If the device uses an internal buffer, the buffer can be created when a file is opened, or the buffer can be created during a read or write.

Return Value

This function must return -1 to indicate an error if for some reason the file could not be opened; such as the file does not exist, could not be created, or there are too many files open. The value of errno may optionally be set to indicate the exact error (the HOST device does not set errno). Some devices might have special failure conditions; for instance, if a device is read-only, a file cannot be opened O_WRONLY.

On success, this function must return a non-negative file descriptor unique among all open files handled by the specific device. The file descriptor need not be unique across devices. The device file descriptor is used only by low-level functions when calling the device-driver-level functions. The low-level function `open` allocates its own unique file descriptor for the high-level functions to call the low-level functions. Code that uses only high-level I/O functions need not be aware of these file descriptors.

DEV_close

Close File for I/O

Syntax

int DEV_close (int *dev_fd* **);**

Description

This function closes a valid open file descriptor.

On some devices, DEV_close may need to be responsible for checking if this is the last file descriptor pointing to a file that was unlinked. If so, it is responsible for ensuring that the file is actually removed from the device and the resources reclaimed, if appropriate.

Return Value

This function should return -1 to indicate an error if the file descriptor is invalid in some way, such as being out of range or already closed, but this is not required. The user should not call close() with an invalid file descriptor.

DEV_read

Read Characters from a File

Syntax

int DEV_read (int *dev_fd* **, char *** *buf* **, unsigned** *count* **);**

Description

The read function reads *count* bytes from the input file associated with *dev_fd*.

- The *dev_fd* is the number assigned by open to an opened file.
- The *buf* is where the read characters are placed.
- The *count* is the number of characters to read from the file.

Return Value

This function must return -1 to indicate an error if for some reason no bytes could be read from the file. This could be because of an attempt to read from a O_WRONLY file, or for device-specific reasons.

If count is 0, no bytes are read and this function returns 0.

This function returns the number of bytes read, from 0 to count. 0 indicates that EOF was reached before any bytes were read. It is not an error to read less than count bytes; this is common if the are not enough bytes left in the file or the request was larger than an internal device buffer size.

DEV_write

Write Characters to a File

Syntax

int DEV_write (int *dev_fd* **, const char *** *buf* **, unsigned** *count* **);**

Description

This function writes *count* bytes to the output file.

- The *dev_fd* is the number assigned by open to an opened file.
- The *buffer* is where the write characters are placed.
- The *count* is the number of characters to write to the file.

Return Value

This function must return -1 to indicate an error if for some reason no bytes could be written to the file. This could be because of an attempt to read from a O_RDONLY file, or for device-specific reasons.

DEV_lseek

Set File Position Indicator

Syntax

off_t DEV_lseek (int *dev_fd* **, off_t** *offset* **, int** *origin* **);**

Description

This function sets the file's position indicator for this file descriptor as lseek.

If lseek is supported, it should not allow a seek to before the beginning of the file, but it should support seeking past the end of the file. Such seeks do not change the size of the file, but if it is followed by a write, the file size will increase.

Return Value

If successful, this function returns the new value of the file position indicator.

This function must return -1 to indicate an error if for some reason no bytes could be written to the file. For many devices, the lseek operation is nonsensical (e.g. a computer monitor).

DEV_unlink

Delete File

Syntax

int DEV_unlink (const char * *path* **);**

Description

Remove the association of the pathname with the file. This means that the file may no longer be opened using this name, but the file may not actually be immediately removed.

Depending on the device, the file may be immediately removed, but for a device which allows open file descriptors to point to unlinked files, the file will not actually be deleted until the last file descriptor is closed. See Section 9.2.3.

Return Value

This function must return -1 to indicate an error if for some reason the file could not be unlinked (delayed removal does not count as a failure to unlink.)

If successful, this function returns 0.

DEV_rename

Rename File

Syntax

int DEV_rename (const char * *old_name* **, const char *** *new_name* **);**

Description

This function changes the name associated with the file.

- The *old_name* is the current name of the file.
- The *new_name* is the new name for the file.

Return Value

This function must return -1 to indicate an error if for some reason the file could not be renamed, such as the file does not exist, or the new name already exists.

Note

It is inadvisable to allow renaming a file so that it is on a different device. In general this would require a whole file copy, which may be more expensive than you expect.

If successful, this function returns 0.

9.2.4 Adding a User-Defined Device Driver for C I/O

The function add_device allows you to add and use a device. When a device is registered with add_device, the high-level I/O routines can be used for I/O on that device.

You can use a different protocol to communicate with any desired device and install that protocol using add_device; however, the HOST functions should not be modified. The default streams stdin, stdout, and stderr can be remapped to a file on a user-defined device instead of HOST by using freopen() as in Example 9-1. If the default streams are reopened in this way, the buffering mode will change to _IOFBF (fully buffered). To restore the default buffering behavior, call setvbuf on each reopened file with the appropriate value (_IOLBF for stdin and stdout, _IONBF for stderr).

The default streams stdin, stdout, and stderr can be mapped to a file on a user-defined device instead of HOST by using freopen() as shown in Example 9-1. Each function must set up and maintain its own data structures as needed. Some function definitions perform no action and should just return.

Note

Use Unique Function Names

The function names open, read, write, close, lseek, rename, and unlink are used by the low-level routines. Use other names for the device-level functions that you write.

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 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.

Example 9-1. Mapping Default Streams to Device

```
#include <stdio.h>
#include <file.h>
#include "mydevice.h"
void main()
{
    add_device("mydevice", _MSA,
               MYDEVICE_open, MYDEVICE_close,
               MYDEVICE_read, MYDEVICE_write,
               MYDEVICE_lseek, MYDEVICE_unlink, MYDEVICE_rename);
    /*--------------------------------------------------------------------*/
    /* Re-open stderr as a MYDEVICE file                                  */
    /*--------------------------------------------------------------------*/
    if (!freopen("mydevice:stderrfile", "w", stderr))
    {
        puts("Failed to freopen stderr");
        exit(EXIT_FAILURE);
    }
    /*--------------------------------------------------------------------*/
    /* stderr should not be fully buffered; we want errors to be seen as  */
    /* soon as possible.  Normally stderr is line-buffered, but this example */
    /* does not buffer stderr at all. This means that there will be one call */
    /* to write() for each character in the message.                      */
    /*--------------------------------------------------------------------*/
    if (setvbuf(stderr, NULL, _IONBF, 0))
    {
        puts("Failed to setvbuf stderr");
        exit(EXIT_FAILURE);
    }
    /*--------------------------------------------------------------------*/
    /* Try it out!                                                        */
    /*--------------------------------------------------------------------*/
    printf("This goes to stdout\n");
    fprintf(stderr, "This goes to stderr\n"); }
```

9.2.5 The device Prefix

A file can be opened to a user-defined device driver by using a device prefix in the pathname. The device prefix is the device name used in the call to add_device followed by a colon. For example:

```
FILE *fptr = fopen("mydevice:file1", "r");
int fd = open("mydevice:file2, O_RDONLY, 0);
```

If no device prefix is used, the HOST device will be used to open the file.

add_device

Add Device to Device Table

Syntax for C

#include <file.h>

int add_device(char * *name*,
unsigned *flags* **,**
int (* *dopen* **)(const char *** *path* **, unsigned** *flags* **, int llv_fd),**
int (* *dclose* **)(int dev_fd),**
int (* *dread* **)(int** *dev_fd* **, char *** *buf* **, unsigned** *count* **),**
int (* *dwrite* **)(int** *dev_fd* **, const char *** *buf* **, unsigned** *count* **),**
off_t (* *dlseek* **)(int dev_fd, off_t** *ioffset* **, int** *origin* **),**
int (* *dunlink* **)(const char *** *path* **),**
int (* *drename* **)(const char *** *old_name* **, const char *** *new_name* **));**

Defined in

lowlev.c (in the lib/src subdirectory of the compiler installation)

Description

The add_device function adds a device record to the device table allowing that device to be used for I/O 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 file.h.
- The *dopen*, *dclose*, *dread*, *dwrite*, *dlseek*, *dunlink*, and *drename* specifiers are function pointers to the functions in the device driver that are called by the low-level functions to perform I/O on the specified device. You must declare these functions with the interface specified in Section 9.2.2. The device driver 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	on failure

add_device (continued)

Add Device to Device Table

Example

Example 9-2 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 pointer *fid*
- Writes the string *Hello, world* into the file
- Closes the file

Example 9-2 illustrates adding and using a device for C I/O:

Example 9-2. Program for C I/O Device

```
#include <file.h>
#include <stdio.h>
/*****************************************************************************/
/* Declarations of the user-defined device drivers                           */
/*****************************************************************************/
extern int   MYDEVICE_open(const char *path, unsigned flags, int fno);
extern int   MYDEVICE_close(int fno);
extern int   MYDEVICE_read(int fno, char *buffer, unsigned count);
extern int   MYDEVICE_write(int fno, const char *buffer, unsigned count);
extern off_t MYDEVICE_lseek(int fno, off_t offset, int origin);
extern int   MYDEVICE_unlink(const char *path);
extern int   MYDEVICE_rename(const char *old_name, char *new_name);
main()
{
   FILE *fid;
   add_device("mydevice", _MSA, MYDEVICE_open, MYDEVICE_close, MYDEVICE_read,
              MYDEVICE_write, MYDEVICE_lseek, MYDEVICE_unlink, MYDEVICE_rename);
   fid = fopen("mydevice:test","w");
   fprintf(fid,"Hello, world\n");

   fclose(fid);
}
```

9.3 Handling Reentrancy (_register_lock() and _register_unlock() Functions)

The C standard assumes only one thread of execution, with the only exception being extremely 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, SYS/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 SYS/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 release the semaphore.

Usually SYS/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 that do not use the SYS/BIOS locking 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.

9.4 Library-Build Process

When using the C/C++ compiler, you can compile your code under a large number of different configurations and options that are not necessarily compatible with one another. Because it would be infeasible to include all possible run-time-support library variants, compiler releases pre-build only a small number of very commonly-used libraries such as rts64plus.lib.

To provide maximum flexibility, the run-time-support source code is provided as part of each compiler release. You can build the missing libraries as desired. The linker can also automatically build missing libraries. This is accomplished with a new library build process, the core of which is the executable mklib, which is available beginning with CCS 5.1.

9.4.1 Required Non-Texas Instruments Software

To use the self-contained run-time-support build process to rebuild a library with custom options, the following are required:

- sh (Bourne shell)
- gmake (GNU make 3.81 or later)

  More information is available from GNU at http://www.gnu.org/software/make. GNU make (gmake) is also available in earlier versions of Code Composer Studio. 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 build for Windows32" when the following is executed from the Command Prompt window:

```
gmake -h
```

All three of these programs are provided as a non-optional feature of CCS 5.1. They are also available as part of the optional XDC Tools feature if you are using an earlier version of CCS.

The mklib program looks for these executables in the following order:

1. in your PATH
2. in the directory getenv("CCS_UTILS_DIR")/cygwin
3. in the directory getenv("CCS_UTILS_DIR")/bin
4. in the directory getenv("XDCROOT")
5. in the directory getenv("XDCROOT")/bin

If you are invoking mklib from the command line, and these executables are not in your path, you must set the environment variable CCS_UTILS_DIR such that getenv("CCS_UTILS_DIR")/bin contains the correct programs.

9.4.2 Using the Library-Build Process

You should normally let the linker automatically rebuild libraries as needed. If necessary, you can run mklib directly to populate libraries. See Section 9.4.2.2 for situations when you might want to do this.

9.4.2.1 Automatic Standard Library Rebuilding by the Linker

The linker looks for run-time-support libraries primarily through the C6X_C_DIR environment variable. Typically, one of the pathnames in C6X_C_DIR is *your install directory*/lib, which contains all of the pre-built libraries, as well as the index library libc.a. The linker looks in C6X_C_DIR to find a library that is the best match for the build attributes of the application. The build attributes are set indirectly according to the command-line options used to build the application. Build attributes include things like CPU revision. If the library name is explicitly specified (e.g. -library=rts64plus.lib), run-time support looks for that library exactly. If the library name is not specified, the linker uses the index library libc.a to pick an appropriate library. If the library is specified by path (e.g. –library=/foo/rts64plus.lib), it is assumed the library already exists and it will not be built automatically.

The index library describes a set of libraries with different build attributes. The linker will compare the build attributes for each potential library with the build attributes of the application and will pick the best fit. For details on the index library, see the archiver chapter in the *TMS320C6000 Assembly Language Tools User's Guide*.

Now that the linker has decided which library to use, it checks whether the run-time-support library is present in C6X_C_DIR. The library must be in exactly the same directory as the index library libc.a. If the library is not present, the linker invokes mklib to build it. This happens when the library is missing, regardless of whether the user specified the name of the library directly or allowed the linker to pick the best library from the index library.

The mklib program builds the requested library and places it in 'lib' directory part of C6X_C_DIR in the same directory as the index library, so it is available for subsequent compilations.

Things to watch out for:

- The linker invokes **mklib** and waits for it to finish before finishing the link, so you will experience a one-time delay when an uncommonly-used library is built for the first time. Build times of 1-5 minutes have been observed. This depends on the power of the host (number of CPUs, etc).
- In a shared installation, where an installation of the compiler is shared among more than one user, it is possible that two users might cause the linker to rebuild the same library at the same time. The **mklib** program tries to minimize the race condition, but it is possible one build will corrupt the other. In a shared environment, all libraries which might be needed should be built at install time; see Section 9.4.2.2 for instructions on invoking **mklib** directly to avoid this problem.
- The index library must exist, or the linker is unable to rebuild libraries automatically.
- The index library must be in a user-writable directory, or the library is not built. If the compiler installation must be installed read-only (a good practice for shared installation), any missing libraries must be built at installation time by invoking **mklib** directly.
- The **mklib** program is specific to a certain version of a certain library; you cannot use one compiler version's run-time support's **mklib** to build a different compiler version's run-time support library.

9.4.2.2 Invoking mklib Manually

You may need to invoke **mklib** directly in special circumstances:

- The compiler installation directory is read-only or shared.
- You want to build a variant of the run-time-support library that is not pre-configured in the index library **libc.a** or known to mklib. (e.g. a variant with source-level debugging turned on.)

9.4.2.2.1 Building Standard Libraries

You can invoke mklib directly to build any or all of the libraries indexed in the index library **libc.a**. The libraries are built with the standard options for that library; the library names and the appropriate standard option sets are known to mklib.

This is most easily done by changing the working directory to be the compiler run-time-support library directory 'lib' and invoking the **mklib** executable there:

```
mklib --pattern=rts64plus.lib
```

9.4.2.2.2 Shared or Read-Only Library Directory

If the compiler tools are to be installed in shared or read-only directory, mklib cannot build the standard libraries at link time; the libraries must be built before the library directory is made shared or read-only.

At installation time, the installing user must build all of the libraries which will be used by any user. To build all possible libraries, change the working directory to be the compiler RTS library directory 'lib' and invoke the mklib executable there:

```
mklib --all
```

Some targets have many libraries, so this step can take a long time. To build a subset of the libraries, invoke mklib individually for each desired library.

9.4.2.2.3 Building Libraries With Custom Options

You can build a library with any extra custom options desired. This is useful for building a version of the library with silicon exception workarounds enabled. The generated library is not a standard library, and must not be placed in the 'lib' directory. It should be placed in a directory local to the project which needs it. To build a

debugging version of the library rts64plus.lib, change the working directory to the 'lib' directory and run the command:

```
mklib --pattern=rts64plus.lib --name=rts64plus_debug.lib --install_to=$Project/Debug --
extra_options="-g"
```

9.4.2.2.4 The mklib Program Option Summary

Run the following command to see the full list of options. These are described in Table 9-2.

```
mklib --help
```

Table 9-2. The mklib Program Options

Option	Effect
--index= *filename*	The index library (libc.a) for this release. Used to find a template library for custom builds, and to find the source files (in the lib/src subdirectory of the compiler installation). REQUIRED.
--pattern= *filename*	Pattern for building a library. If neither --extra_options nor --options are specified, the library will be the standard library with the standard options for that library. If either --extra_options or --options are specified, the library is a custom library with custom options. REQUIRED unless --all is used.
--all	Build all standard libraries at once.
--install_to= *directory*	The directory into which to write the library. For a standard library, this defaults to the same directory as the index library (libc.a). For a custom library, this option is REQUIRED.
--compiler_bin_dir= *directory*	The directory where the compiler executables are. When invoking mklib directly, the executables should be in the path, but if they are not, this option must be used to tell mklib where they are. This option is primarily for use when mklib is invoked by the linker.
--name= *filename*	File name for the library with no directory part. Only useful for custom libraries.
--options=' *str* **'**	Options to use when building the library. The default options (see below) are *replaced* by this string. If this option is used, the library will be a custom library.
--extra_options=' *str* **'**	Options to use when building the library. The default options (see below) are also used. If this option is used, the library will be a custom library.
--list_libraries	List the libraries this script is capable of building and exit. ordinary system-specific directory.
--log= *filename*	Save the build log as *filename*.
--tmpdir= *directory*	Use *directory* for scratch space instead of the ordinary system-specific directory.
--gmake= *filename*	Gmake-compatible program to invoke instead of "gmake"
--parallel= *N*	Compile *N* files at once ("gmake -j N").
--query= *filename*	Does this script know how to build FILENAME?
--help or **--h**	Display this help.
--quiet or **--q**	Operate silently.
--verbose or **--v**	Extra information to debug this executable.

Examples:

To build all standard libraries and place them in the compiler's library directory:

```
mklib --all --index=$C_DIR/lib
```

To build one standard library and place it in the compiler's library directory:

```
mklib --pattern=rts64plus.lib --index=$C_DIR/lib
```

To build a custom library that is just like rts64plus.lib, but has symbolic debugging support enabled:

```
mklib --pattern=rts64plus.lib --extra_options="-g" --index=$C_DIR/lib --install_to=$Project/Debug --
name=rts64plus_debug.lib
```

9.4.3 Extending mklib

The **mklib** API is a uniform interface that allows Code Composer Studio to build libraries without needing to know exactly what underlying mechanism is used to build it. Each library vendor (e.g. the TI compiler) provides a library-specific copy of 'mklib' in the library directory that can be invoked, which understands a standardized set of options, and understands how to build the library. This allows the linker to automatically build application-compatible versions of any vendor's library without needing to register the library in advance, as long as the vendor supports mklib.

9.4.3.1 Underlying Mechanism

The underlying mechanism can be anything the vendor desires. For the compiler run-time-support libraries, mklib is just a wrapper that knows how to use the files in the lib/src subdirectory of the compiler installation and invoke gmake with the appropriate options to build each library. If necessary, mklib can be bypassed and the Makefile used directly, but this mode of operation is not supported by TI, and you are responsible for any changes to the Makefile. The format of the Makefile and the interface between mklib and the Makefile is subject to change without notice. The mklib program is the forward-compatible path.

9.4.3.2 Libraries From Other Vendors

Any vendor who wishes to distribute a library that can be rebuilt automatically by the linker must provide:

- An index library (like 'libc.a', but with a different name)
- A copy of mklib specific to that library
- A copy of the library source code (in whatever format is convenient)

These things must be placed together in one directory that is part of the linker's library search path (specified either in C6X_C_DIR or with the linker --search_path option).

If mklib needs extra information that is not possible to pass as command-line options to the compiler, the vendor will need to provide some other means of discovering the information (such as a configuration file written by a wizard run from inside CCS).

The vendor-supplied mklib must at least accept all of the options listed in Table 9-2 without error, even if they do not do anything.

Chapter 10

C++ Name Demangler

The C++ compiler implements function overloading, operator overloading, and type-safe linking by encoding a function's prototype and namespace in its link-level name. The process of encoding the prototype into the linkname is often referred to as name mangling. When you inspect mangled names, such as in assembly files, disassembler output, or compiler or linker diagnostic messages, 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.

10.1 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.
filenames	Text input files, such as the assembly file output by the compiler, the assembler listing file, the disassembly file, and the linker map file. If no filenames are specified on the command line, dem6x uses standard input.

By default, the C++ name demangler outputs to standard output. You can use the -o file option if you want to output to a file.

The following options apply only to the C++ name demangler:

--debug (--d)	Prints debug messages.
--diag_wrap[=on,off]	Sets diagnostic messages to wrap at 79 columns (on, which is the default) or not (off).
--help (-h)	Prints a help screen that provides an online summary of the C++ name demangler options.
--output= *file* (-o)	Outputs to the specified file rather than to standard out.
--quiet (-q)	Reduces the number of messages generated during execution.
-u	Specifies that external names do not have a C++ prefix. (deprecated)

10.2 Sample Usage of the C++ Name Demangler

The examples in this section illustrate the demangling process.

This example shows a sample C++ program. In this example, the linknames of all the functions are mangled; that is, their signature information is encoded into their names.

```
class banana {
public:
    int calories(void);
    banana();
    ~banana();
};
int calories_in_a_banana(void)
{
    banana x;
    return x.calories();
}
```

The resulting assembly that is output by the compiler is as follows.

```
_calories_in_a_banana__Fv:
;** --------------------------------------------------------------------------*
           CALL    .S1     ___ct__6bananaFv
           STW     .D2T2   B3,*SP--(16)
           MVKL    .S2     RL0,B3
           MVKH    .S2     RL0,B3
           ADD     .S1X    8,SP,A4
           NOP             1
RL0:       ; CALL OCCURS
           CALL    .S1     _calories__6bananaFv
           MVKL    .S2     RL1,B3
           ADD     .S1X    8,SP,A4
           MVKH    .S2     RL1,B3
           NOP             2
RL1:       ; CALL OCCURS
           CALL    .S1     ___dt__6bananaFv
           STW     .D2T1   A4,*+SP(4)
           ADD     .S1X    8,SP,A4
           MVKL    .S2     RL2,B3
           MVK     .S2     0x2,B4
           MVKH    .S2     RL2,B3
RL2:       ; CALL OCCURS
           LDW     .D2T1   *+SP(4),A4
           LDW     .D2T2   *++SP(16),B3
           NOP             4
           RET     .S2     B3
           NOP             5
           ; BRANCH OCCURS
```

Executing the C++ name demangler will demangle all names that it believes to be mangled. Enter:

```
dem6x calories_in_a_banana.asm
```

The result after running the C++ name demangler is as follows. The linknames in _ZN6bananaC1Ev, _ZN6banana8caloriesEv, and _ZN6bananaD1Ev are demangled.

```
calories_in_a_banana():
;** --------------------------------------------------------------------------*
        CALL    .S1 banana::banana()
        STW     .D2T2 B3,*SP--(16)
        MVKL    .S2 RL0,B3
        MVKH    .S2 RL0,B3
        ADD     .S1X 8,SP,A4
        NOP 1
RL0:    ; CALL OCCURS
        CALL    .S1 banana::calories()
        MVKL    .S2 RL1,B3
        ADD .   S1X 8,SP,A4
        MVKH    .S2 RL1,B3
        NOP 2
RL1:    ; CALL OCCURS
        CALL    .S1 banana::~banana()
        STW     .D2T1 A4,*+SP(4)
        ADD     .S1X 8,SP,A4
        MVKL    .S2 RL2,B3
        MVK .   S2 0x2,B4
        MVKH .  S2 RL2,B3
RL2:    ; CALL OCCURS
        LDW     .D2T1 *+SP(4),A4
        LDW     .D2T2 *++SP(16),B3
        NOP 4
        RET .S2 B3
        NOP 5
        ; BRANCH OCCURS
```

This page intentionally left blank.

Appendix A
Glossary

A.1 Terminology

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.

Application Binary Interface (ABI) A standard that specifies the interface between two object modules. An ABI specifies how functions are called and how information is passed from one program component to another.

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.

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. This ABI is no longer supported.

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.

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 Linkable 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.
executable object file	A linked, executable object file that is downloaded and executed on 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.
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.
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 object file.
input section	A section from an object file that will be linked into an executable object file.
integrated preprocessor	A C/C++ preprocessor that is merged with the parser, allowing for faster compilation. Stand-alone preprocessing or preprocessed listing is also available.
interlist feature	A feature that inserts as comments your original C/C++ source statements into the assembly language output from the assembler. The C/C++ statements are inserted next to the equivalent assembly instructions.
intrinsics	Operators that are used like functions and produce assembly language code that would otherwise be inexpressible in C, or would take greater time and effort to code.
ISO	International Organization for Standardization; a worldwide federation of national standards bodies, which establishes international standards voluntarily followed by industries.

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.

linker A software program that combines object files to form an executable object file that can be allocated into system memory and executed by the device.

listing file An output file, created by the assembler, which 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*

loader A device that places an executable object file 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, which 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.

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 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.
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.
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.
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, which 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.

sign extend	A process that fills the unused MSBs of a value with the value's sign bit.
software pipelining	A technique used by the C/C++ optimizer to schedule instructions from a loop so that multiple iterations of the loop execute in parallel.
source file	A file that contains C/C++ code or assembly language code that is compiled or assembled to form an object file.
stand-alone preprocessor	A software tool that expands macros, #include files, and conditional compilation as an independent program. It also performs integrated preprocessing, which includes parsing of instructions.
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.
string table	A table that stores symbol names that are longer than eight characters (symbol names of eight characters or longer cannot be stored in the symbol table; instead they are stored in the string table). The name portion of the symbol's entry points to the location of the string in the string table.
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.
symbolic debugging	The ability of a software tool to retain symbolic information that can be used by a debugging tool such as 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.
word	A 32-bit addressable location in target memory

Revision History

Changes from January 17, 2023 to April 6, 2023 (from Revision E (January 2023) to Revision F (April 2023)) **Page**

- Removed documentation of vector constructor-style syntax that is not supported for C6000....192

Changes from January 14, 2022 to January 17, 2023 (from Revision D (January 2022) to Revision E (January 2023)) **Page**

- Added the --assume_control_regs_read option.... 23
- The --strict_compatibility linker option no longer has any effect and has been removed from the documentation.... 29
- Added the --fp_single_precision_constant compiler option.... 32
- Added the --assume_control_regs_read option.... 33
- Documented predefined macros for ptrdiff_t and size_t types.... 38
- Corrected names of the --gen_cross_reference_listing and --asm_cross_reference_listing options wherever they appear.... 46
- Added links to additional documentation to the list of --opt_level optimizations.... 56
- Added information about corresponding --opt_level to certain optimization descriptions.... 95
- Native vector types are now called "TI vector types."....151
- Examples have been updated to use constructors to initialize elements of a vector rather than cast/scalar-widening syntax.... 192
- Examples have been updated to use constructors to initialize elements of a vector rather than cast/scalar-widening syntax.... 194
- Added description of the effects of the --assume_control_regs_read option.... 249

Changes from June 15, 2018 to January 14, 2022 (from Revision C (June 2018) to Revision D (January 2022)) **Page**

- This document revision is an update for the v8.3.x release. The previous revision, SPRU514C, was also published for v8.3.x....11
- Updated the numbering format for tables, figures, and cross-references throughout the document....11
- Removed references to the Processors wiki throughout the document.... 11
- Dynamic linking and dynamic loading are no longer supported by the C6000 compiler and linker....23
- The --small_enum option is deprecated.... 23
- MISRA-C checking is no longer supported....23
- Removed documentation of the --rom linker option, which is not supported....29
- MISRA-C checking is no longer supported....32
- The --small_enum option is deprecated.... 33
- Added information about default for --gen_func_subsections option.... 138
- The SET_DATA_SECTION pragma takes precedence over the --gen_data_subsections=on option.... 138
- Corrected information about default for --gen_data_subsections option....138
- Documented that C11 atomic operations are not supported.... 144
- Updated information about the size of enum types.... 150
- Added information about character sets and file encoding....152
- Pragmas for MISRA-C checking are no longer supported.... 161
- Clarify interaction between --opt_level and FUNCTION_OPTIONS pragma.... 172

This document describes the v8.x TI Code Generation Tools for the C64x+, C6740, and C6600 variants of the TMS320C6000™ processor series. The C6200, C6400, C6700, and C6700+ variants are not supported in v8.0 and later versions of the TI Code Generation Tools. In addition, the v8.x tools no longer support the COFF object file format and the associated STABS debugging format. If you are using a legacy device or want COFF support, please use v7.4 of the Code Generation Tools and refer to SPRU187 and SPRU186 for documentation.

The following table lists changes made to this document prior to changes to the document numbering format. The left column identifies the first version of this document in which that particular change appeared.

Table 13-1. Revision History

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPRUI04C	Using the Compiler, C/C++ Language	Section 3.3 and Section 7.2	The compiler now follows the C++14 standard. In addition, removed several older C++ features from the exception list because they have been supported for several releases.
SPRUI04C	Using the Compiler and Optimization	Table 3-12 and Section 4.6.1	Corrected the spelling of the --disable_software_pipeline option.
SPRUI04C	Using the Compiler	Section 3.3.1	Added the --ecc=on linker option, which enables ECC generation. Note that ECC generation is now off by default. Also added the --no_const_clink options, which prevents the compiler from generating .clink directives for const global arrays.
SPRUI04C	Using the Compiler, C/C++ Language	Section 3.11 and Section 7.9	Revised the section on inline function expansion and its subsections to include new pragmas and changes to the compilers decision-making about what functions to inline. The FORCEINLINE, FORCEINLINE_RECURSIVE, and NOINLINE pragmas have been added.
SPRUI04C	C/C++ Language	Section 7.3.2 and Section 7.15	Support for vector data types no longer requires the use of the optimizer.
SPRUI04C	C/C++ Language	Section 7.14.2 and Section 7.14.4	Added "retain" as a function attribute and variable attribute.
SPRUI04C	C/C++ Language	Section 7.15.6	Vector data types can be used with printf() as described in the OpenCL 1.2 specification.
SPRUI04C	Run-Time Environment	Section 8.6.2	Clarified information about the alignment of __x128_t objects.
SPRUI04C	Run-Time Support Functions	Section 9.2.1.1	Added information about the FORMAT_CONVERSION_BUFSIZE macro defined in format.h and used by functions such as printf().
SPRUI04B	Using the Compiler		Several compiler options have been deprecated, removed, or renamed. The compiler continues to accept some of the deprecated options, but they are not recommended for use.
SPRUI04B	Using the Compiler	Section 3.3	The --multithread option has been added as both a compiler and linker option.
SPRUI04B	Using the Compiler, C/C++ Language	Section 3.3.3	Revised to state that --check_misra option is required even if the CHECK_MISRA pragma is used.
SPRUI04B	Using the Compiler	Section 3.10	Corrected the document to describe the ---gen_preprocessor_listing option. The name --gen_parser_listing was incorrect.

Table 13-1. Revision History (continued)

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPRUI04B	Optimization	Section 4.10.3	Corrected function names for _TI_start_pprof_collection() and _TI_stop_pprof_collection().
SPRUI04B	Linking C/C++ Code	Section 6.3.5	Added .ovly and .TI.crctab to list of initialized sections created by the compiler.
SPRUI04B	Run-Time Environment	Section 8.6.6	Identified the intrinsics that are defined as macros and so require that the c6x.h header file be included.
SPRUI04A	Using the Compiler	Section 3.3 and Section 6.2.3	The --gen_data_subsections option has been added.
SPRUI04A	Run-Time Environment	Section 8.9.1	Additional boot hook functions are available. These can be customized for use during system initialization.
SPRUI04	Introduction	Section 1.4	The COFF object file format and the associated STABS debugging format are no longer supported. The C6000 compiler now supports only the Embedded Application Binary Interface (EABI) ABI, which works only with object files that use the ELF object file format and the DWARF debug format. Sections of this document that referred to the COFF format have been removed or simplified. If you need COFF support, please use v7.4 of the Code Generation Tools and refer to SPRU187 and SPRU186 for documentation.
SPRUI04	Getting Started	Chapter 2	Added a Getting Started chapter with introductory information for new users.
SPRUI04	Using the Compiler	Section 3.3.4	The --ramfunc option has been added to the compiler command-line options.
SPRUI04	Using the Compiler	Section 3.3.5	The C6200, C6400, C6700, and C6700+ variants are no longer supported. Sections of this document that referred to these devices have been removed or simplified. If you are using one of these legacy devices, please use v7.4 of the Code Generation Tools and refer to SPRU187 and SPRU186 for documentation.
SPRUI04	Using the Compiler	Section 3.6	Added section on techniques for passing arguments to main().
SPRUI04	Using the Compiler	Section 3.11	Corrected to state that automatic inlining is performed with --opt_level=3, but not --opt_level=2.
SPRUI04	C/C++ Language	Section 7.9.7	Added the diag_push and diag_pop diagnostic message pragmas.
SPRUI04	C/C++ Language	Section 7.9.24	Added the NOINIT and PERSISTENT pragmas.
SPRUI04	C/C++ Language	Section 7.14.2	The ramfunc function attribute has been added.
SPRUI04	C/C++ Language	Section 7.3.2 and Section 7.15	Added vector data types.
SPRUI04	Run-Time Environment	Section 8.5	Added reference to section on accessing linker symbols in C and C++ in the *Assembly Language Tools User's Guide*.
SPRUI04	Run-Time Environment	Section 8.6.6	Corrected the operands for the _shr2 and _shru2 intrinsics.

This page intentionally left blank.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

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