TMS320C28x Optimizing C/C++ Compiler v21.6.0.LTS

User's Guide

Literature Number: SPRU514W
AUGUST 2001 – REVISED JUNE 2021

Table of Contents

List of Figures

List of Tables

About This Manual

The *TMS320C28x Optimizing C/C++ Compiler User's Guide* explains how to use the following Texas Instruments Code Generation compiler tools:

- Compiler
- Post-link 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 2003 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:

cl2000 [*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:

cl2000 --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*].
- The TMS320C2800™ core is referred to as TMS320C28x or C28x.

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 14882-2003, International Standard - Programming Languages - C++ (The 2003 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/)

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:

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

SPRU513 ***TMS320C28x Assembly Language Tools User's Guide*** describes the assembly language tools (assembler and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS320C28x device.

SPRU430 ***TMS320C28x DSP CPU and Instruction Set Reference Guide*** describes the central processing unit (CPU) and the assembly language instructions of the TMS320C28x fixed-point digital signal processors (DSPs). It also describes emulation features available on these DSPs.

SPRU566 ***TMS320x28xx, 28xxx Peripheral Reference Guide*** describes the peripheral reference guides of the 28x digital signal processors (DSPs).

SPRUHS1 ***TMS320C28x Extended Instruction Sets Technical Reference Manual*** describes the architecture, pipeline, and instruction sets of the TMU, VCRC, VCU-II, FPU32, and FPU64 accelerators.

SPRAC71 ***TMS320C28x Embedded Application Binary Interface (EABI) Application Report***. Provides a specification for the ELF-based Embedded Application Binary Interface (EABI) for the TMS320C28x 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.

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

TMS320C2800™, TMS320C28x™, and Code Composer Studio™ are trademarks of Texas Instruments.
All trademarks are the property of their respective owners.

Chapter 1

Introduction to the Software Development Tools

The TMS320C28x™ is supported by a set of software development tools, which includes an optimizing C/C++ compiler, 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 *TMS320C28x 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.

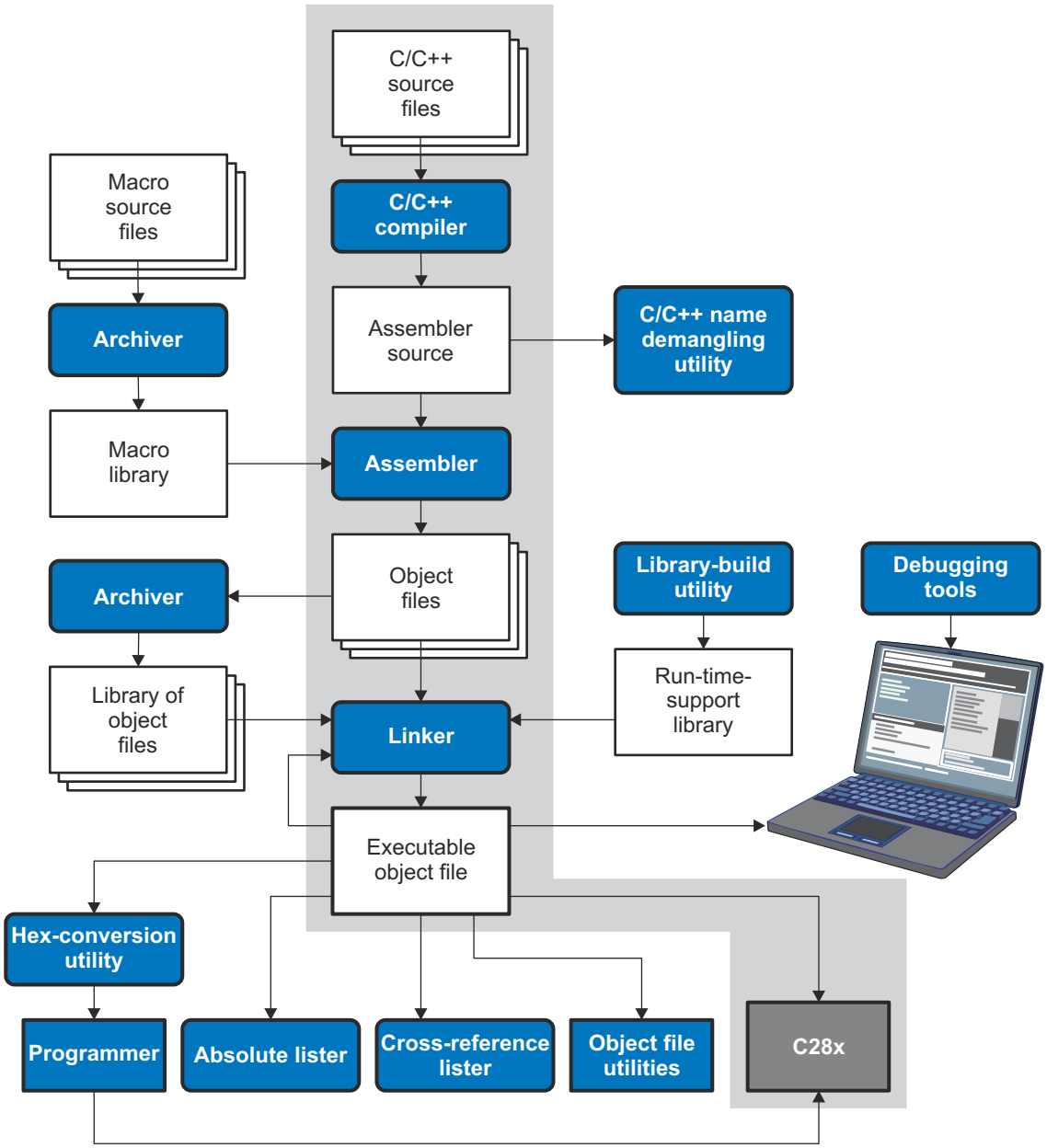


Figure 1-1. TMS320C28x Software Development Flow

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

- The **compiler** accepts C/C++ source code and produces C28x assembly language source code. See Chapter 2.
- The **assembler** translates assembly language source files into machine language relocatable object files. See the *TMS320C28x 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 4 for an overview of the linker. See the *TMS320C28x 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 *TMS320C28x 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 8.

  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 8.5. 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 *TMS320C28x Assembly Language Tools User's Guide*.
- The **absolute lister** accepts linked object files as input and creates .abs files as output. You can assemble these .abs files to produce a listing that contains absolute, rather than relative, addresses. Without the absolute lister, producing such a listing would be tedious and would require many manual operations. See the *TMS320C28x Assembly Language Tools User's Guide*.
- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files. See the *TMS320C28x 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 9.
- The **post-link optimizer** removes or modifies assembly language instructions to generate better code. The post-link optimizer must be run with the compiler -plink option. See Chapter 5.
- The **disassembler** decodes object files to show the assembly instructions that they represent. See the *TMS320C28x Assembly Language Tools User's Guide*.
- The main product of this development process is an executable object file that can be executed on a **TMS320C28x** device.

1.2 Compiler Interface

The compiler is a command-line program named cl2000. 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 2.1

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 7.

1.3 ANSI/ISO Standard

The compiler supports the 1989, 1999, and 2011 versions of the C language and the 2003 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++03 version of the C++ standard. See the C++ Standard ISO/IEC 14882:2003. The language is also described in Ellis and Stroustrup's *The Annotated C++ Reference Manual* (ARM), but that is not the standard. For a description of *unsupported* C++ features, see Section 6.2.
- **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 8.

See Section 6.14 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:

- **COFF object files.** Common object file format (COFF) provides basic modular (separately-compiled) compilation features, such as relocations.
- **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 C28x is extended by the C28x Embedded Application Binary Interface (EABI), which is documented in SPRAC71.

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 8.5.
- **C++ name demangler**

  The C++ name demangler (dem2000) 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 9.
- **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 COFF or ELF files output by the compiler can be converted to EPROM programmer data files by using the hex conversion utility, as described in the *TMS320C28x Assembly Language Tools User's Guide*.

Chapter 2

Using the C/C++ Compiler

The compiler translates your source program into machine language object code that the TMS320C28x 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.

2.1 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 and 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 2.3.9 for more information.
- The **linker** combines object files to create an executable or relinkable file. The link step is optional, so you can compile and assemble many modules independently and link them later. See Chapter 4 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 4.1.1 for details.

For a complete description of the assembler and the linker, see the *TMS320C28x Assembly Language Tools User's Guide*.

2.2 Invoking the C/C++ Compiler

To invoke the compiler, enter:

cl2000 [*options*] [*filenames*] [**--run_linker** [*link_options*] *object files*]]

cl2000	Command that runs the compiler and the assembler.
options	Options that affect the way the compiler processes input files. The options are listed in Table 2-6 through Table 2-28.
filenames	One or more C/C++ source files and assembly language source files.
--run_linker (**-z**)	Option that invokes the linker. The --run_linker option's short form is -z. See Chapter 4 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:

```
cl2000 symtab.c file.c seek.asm --run_linker --library=lnk.cmd
       --output_file=myprogram.out
```

2.3 Changing the Compiler's Behavior with Options

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

For a help screen summary of the options, enter **cl2000** 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 C2000_C_OPTION environment variable. For a detailed description of the environment variable, see Section 2.4.1.

Table 2-6 through Table 2-28 summarize all options (including link options). Use the references in the tables for more complete descriptions of the options.

Table 2-1. Processor Options

Option	Alias	Effect	Section
--silicon_version=28	-v28	Specifies TMS320C28x architecture. The default (and only value accepted) is 28. This option is no longer required.	Section 2.3.4
--abi={coffabi|eabi}		Selects application binary interface. Default is coffabi. Support is also provided for eabi.	Section 2.3.4
--cla_support[=cla0|cla1|cla2]		Specifies TMS320C28x CLA accelerator support for Type 0, Type 1, or Type 2. Default is cla0. Use this option only if the target hardware provides this functionality.	Section 2.3.4
--float_support={ fpu32 | fpu64 | softlib }		Specifies use of TMS320C28x 32-bit or 64-bit hardware floating-point support. The default is softlib. Use this option only if the target hardware provides this functionality.	Section 2.3.4
--idiv_support={none | idiv0}		Enables support for fast integer division using hardware extensions to provide a set of instructions to accelerate integral division for 16-, 32-, and 64-bit values. If this hardware is available, use --idiv_support=idiv0 to cause these instructions to be used. Use this option only if the target hardware provides this functionality. The default is none. (EABI only)	Section 2.3.4
--lfu_reference_elf=*path*	-lfu=*path*	In order to create a Live Firmware Update (LFU) compatible executable binary, specify the path to a previous ELF executable binary to use as a reference from which to obtain a list of the memory addresses of global and static symbols. This previous binary may be an LFU-compatible binary, but this is not required. (LFU is supported for EABI only)	Section 2.15

Table 2-1. Processor Options (continued)

Option	Alias	Effect	Section
-lfu_default[=none | preserve]		Specify the default treatment of global and static symbol addresses found in the reference ELF executable if they do not have the update or preserve attribute in the new executable. These treatments are used during a live firmware upate ("warm start"). If --lfu_default=preserve (the default), the compiler preserves all global and static symbol addresses found in the reference ELF executable unless __attribute__((update)) is specified for a symbol. If --lfu_default=none, the compiler preserves only the addresses of symbols that have __attribute__((preserve)) specified. It re-initializes symbols that have __attribute__((update)) specified. All other global and static symbols can be allocated to any memory address by the linker, but are not re-initialized when a warm start occurs. (LFU is supported for EABI only)	Section 2.15
--silicon_errata_fpu1_workaround =on|off		Enabling this option prevents FPU register write conflicts that can occur during certain instructions. The compiler adds NOP instructions before such instructions to prevent conflicts. Use this option only if the target hardware provides FPU functionality.	Section 2.3.4
--tmu_support[=tmu0 | tmu1]		Enables support for the Trigonometric Math Unit (TMU). Using this option also enables FPU32 support (as with --float_support=fpu32). If this option is used but no value is specified, the default is tmu0. The tmu1 option enables support for all tmu0 functionality plus the LOG2F32 and IEXP2F32 instructions. Use this option only if the target hardware provides this functionality. (TMU1 is supported for EABI only.)	Section 2.3.4
--vcu_support[=vcu0|vcu2|vcrc]		Specifies C28x VCU coprocessor support Type 0, Type 2, or VCRC. Use this option only if the target hardware provides this functionality. Default is vcu0.	Section 2.3.4
--unified_memory	-mt	Generates code for the unified memory model.	Section 2.3.4

Table 2-2. Optimization Options(1)

Option	Alias	Effect	Section
--opt_level=off		Disables all optimization (default).	Section 3.1
--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. Level 4 (-O4) uses Level 3 optimizations and performs link-time optimization.	Section 3.1, Section 3.3 , Section 3.6
--opt_for_space=*n*	-ms	Controls code size on four levels (0, 1, 2, and 3).	Section 3.13
--opt_for_speed[=*n*]	-mf	Controls the tradeoff between size and speed (0-5 range). If this option is specified without *n*, the default value is 4. If this option is not specified, the default setting is 2.	Section 3.2

(1) **Note:** Machine-specific options (see Table 2-12) can also affect optimization.

Table 2-3. 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 3.5

Table 2-3. 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 3.4.1
--disable_inlining		Prevents any inlining from occurring.	Section 2.11
--fp_mode={relaxed|strict}		Enables or disables relaxed floating-point mode.	Section 2.3.3
--fp_reassoc={on|off}		Enables or disables the reassociation of floating-point arithmetic.	Section 2.3.3
--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 3.3.1
--isr_save_vcu_regs={on|off}		Generates VCU register save/restore to stack for interrupt routines so that VCU code can be re-entrant.	Section 3.14
--optimizer_interlist	-os	Interlists optimizer comments with assembly statements.	Section 3.10
--program_level_compile	-pm	Combines source files to perform program-level optimization.	Section 3.4
--sat_reassoc={on|off}		Enables or disables the reassociation of saturating arithmetic. Default is --sat_reassoc=off.	Section 2.3.3
--aliased_variables	-ma	Notifies the compiler that addresses passed to functions may be modified by an alias in the called function.	Section 3.9.2.2

(1) **Note:** Machine-specific options (see Table 2-12) can also affect optimization.

Table 2-4. 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 2.3.5 Section 3.12
--symdebug:dwarf_version=2|3|4		Specifies the DWARF format version. The default version is 3 for the COFF ABI and 4 for EABI.	Section 2.3.5
--symdebug:none		Disables all symbolic debugging.	Section 2.3.5 Section 3.12
--symdebug:profile_coff		Enables profiling using the alternate STABS debugging format. STABS is supported only for the COFF ABI.	Section 2.3.5
--symdebug:skeletal		(Deprecated; has no effect.)	

Table 2-5. Include Options

Option	Alias	Effect	Section
--include_path=*directory*	`-I`	Adds the specified directory to the #include search path.	Section 2.5.2.1
--preinclude=*filename*		Includes *filename* at the beginning of compilation.	Section 2.3.3

Table 2-6. Control Options

Option	Alias	Effect	Section
--compile_only	-c	Disables linking (negates --run_linker).	Section 4.1.3
--help	-h	Prints (on the standard output device) a description of the options understood by the compiler.	Section 2.3.2
--run_linker	-z	Causes the linker to be invoked from the compiler command line.	Section 2.3.2

Table 2-6. Control Options (continued)

Option	Alias	Effect	Section
--skip_assembler	-n	Compiles C/C++ source file , producing an assembly language output file. The assembler is not run and no object file is produced.	Section 2.3.2

Table 2-7. Language Options

Option	Alias	Effect	Section
--c89		Processes C files according to the ISO C89 standard.	Section 6.14
--c99		Processes C files according to the ISO C99 standard.	Section 6.14
--c11		Processes C files according to the ISO C11 standard.	Section 6.14
--c++03		Processes C++ files according to the ISO C++03 standard.	Section 6.14
--cla_default		Processes both .c and .cla files as CLA files.	
--cla_signed_compare_workaround ={on|off}		Enables automatic use of floating-point comparisons when compiling CLA comparisons that may result in incorrect answers if integer comparison is used. Off by default.	
--cpp_default	-fg	Processes all source files with a C extension as C++ source files.	Section 2.3.7
--exceptions		Enables C++ exception handling.	Section 6.6
--extern_c_can_throw		Allow extern C functions to propagate exceptions. (EABI only)	--
--float_operations_allowed ={none|all|32|64}		Restricts the types of floating point operations allowed.	Section 2.3.3
--gen_cross_reference	-px	Generates a cross-reference listing file (.crl).	Section 2.9
--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 2.3.4
--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 2.3.3
--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 6.14.3
--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 6.14.3

Table 2-8. 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 2.5.8
--preproc_includes[=*filename*]	-ppi	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive.	Section 2.5.9
--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 2.5.10
--preproc_only	-ppo	Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension.	Section 2.5.4
--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 2.5.6
--preproc_with_compile	-ppa	Continues compilation after preprocessing with any of the -pp<x> options that normally disable compilation.	Section 2.5.5
--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 2.5.7

Table 2-9. Predefined Macro Options

Option	Alias	Effect	Section
--define=*name*[=*def*]	-D	Predefines *name*.	Section 2.3.2
--undefine=*name*	-U	Undefines *name*.	Section 2.3.2

Table 2-10. Diagnostic Message Options

Option	Alias	Effect	Section
--advice:performance[=all,none]		Provides advice on ways to improve performance. Default is all.	Section 2.3.3
--compiler_revision		Prints out the compiler release revision and exits.	--
--diag_error=*num*	-pdse	Categorizes the diagnostic identified by *num* as an error.	Section 2.7.1
--diag_remark=*num*	-pdsr	Categorizes the diagnostic identified by *num* as a remark.	Section 2.7.1
--diag_suppress=*num*	-pds	Suppresses the diagnostic identified by *num*.	Section 2.7.1
--diag_warning=*num*	-pdsw	Categorizes the diagnostic identified by *num* as a warning.	Section 2.7.1
--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 2.7.1
--emit_warnings_as_errors	-pdew	Treat warnings as errors.	Section 2.7.1
--issue_remarks	-pdr	Issues remarks (non-serious warnings).	Section 2.7.1
--no_warnings	-pdw	Suppresses diagnostic warnings (errors are still issued).	Section 2.7.1
--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 2.7.1
--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 2.7.1
--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 2.7.1

Table 2-11. Supplemental Information Options

Option	Alias	Effect	Section
--gen_preprocessor_listing	-pl	Generates a raw listing file (.rl).	Section 2.10

Table 2-12. Run-Time Model Options

Option	Alias	Effect	Section
--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 4.2.2
--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 4.2.1
--no_rpt	-mi	Disables generation of RPT instructions.	Section 2.3.4
--protect_volatile	-mv	Enables volatile reference protection.	Section 2.3.4
--ramfunc={on|off}		If set to on, specifies that all functions should be placed in the .TI.ramfunc section, which is placed in RAM.	Section 2.3.4
--rpt_threshold=*k*		Generates RPT loops that iterate *k* times or less. (k is a constant between 0 and 256.)	Section 2.3.4

Table 2-13. Entry/Exit Hook Options

Option	Alias	Effect	Section
--entry_hook[=*name*]		Enables entry hooks.	Section 2.14
--entry_parm={none|name|address}		Specifies the parameters to the function to the --entry_hook option.	Section 2.14
--exit_hook[=*name*]		Enables exit hooks.	Section 2.14
--exit_parm={none|name|address}		Specifies the parameters to the function to the --exit_hook option.	Section 2.14
--remove_hooks_when_inlining		Removes entry/exit hooks for auto-inlined functions.	Section 2.14

Table 2-14. Feedback Options

Option	Alias	Effect	Section
--analyze=codecov		Generate analysis info from profile data.	Section 3.8.2.2
--analyze_only		Only generate analysis.	Section 3.8.2.2
--gen_profile_info		Generates instrumentation code to collect profile information.	Section 3.7.1.3
--use_profile_info=*file1*[, *file2*,...]		Specifies the profile information file(s).	Section 3.7.1.3

Table 2-15. Assembler Options

Option	Alias	Effect	Section
--keep_asm	-k	Keeps the assembly language (.asm) file.	Section 2.3.11
--asm_listing	-al	Generates an assembly listing file.	Section 2.3.11
--c_src_interlist	-ss	Interlists C source and assembly statements.	Section 2.12 Section 3.10
--src_interlist	-s	Interlists optimizer comments (if available) and assembly source statements; otherwise interlists C and assembly source statements.	Section 2.3.2
--absolute_listing	-aa	Enables absolute listing.	Section 2.3.11
--asm_define=*name*[=*def*]	-ad	Sets the *name* symbol.	Section 2.3.11
--asm_dependency	-apd	Performs preprocessing; lists only assembly dependencies.	Section 2.3.11
--asm_includes	-api	Performs preprocessing; lists only included #include files.	Section 2.3.11
--issue_remarks		Issues remarks (non-serious warnings), which include additional assembly-time checking.	Section 2.3.11
--asm_undefine=*name*	-au	Undefines the predefined constant *name*.	Section 2.3.11
--asm_listing_cross_reference	-ax	Generates the cross-reference file.	Section 2.3.11
--flash_prefetch_warn		Assembler warnings for F281X BF flash prefetch issue.	Section 2.3.11
--include_file=*filename*	-ahi	Includes the specified file for the assembly module.	Section 2.3.11
--preproc_asm	-mx	Preprocesses assembly source, expands assembly macros.	Section 2.3.11

Table 2-16. File Type Specifier Options

Option	Alias	Effect	Section
--asm_file=*filename*	-fa	Identifies *filename* as an assembly source file regardless of its extension. By default, the compiler and assembler treat .asm files as assembly source files.	Section 2.3.7
--c_file=*filename*	-fc	Identifies *filename* as a C source file regardless of its extension. By default, the compiler treats .c files as C source files.	Section 2.3.7
--cpp_file=*filename*	-fp	Identifies *filename* as a C++ file, regardless of its extension. By default, the compiler treats .C, .cpp, .cc and .cxx files as a C++ files.	Section 2.3.7
--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 2.3.7

Table 2-17. Directory Specifier Options

Option	Alias	Effect	Section
--abs_directory=*directory*	-fb	Specifies an absolute listing file directory. By default, the compiler uses the object file directory.	Section 2.3.10

Table 2-17. Directory Specifier Options (continued)

Option	Alias	Effect	Section
--asm_directory=*directory*	-fs	Specifies an assembly file directory. By default, the compiler uses the current directory.	Section 2.3.10
--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 2.3.10
--obj_directory=*directory*	-fr	Specifies an object file directory. By default, the compiler uses the current directory.	Section 2.3.10
--output_file=*filename*	-fe	Specifies a compilation output file name; can override --obj_directory.	Section 2.3.10
--pp_directory=*dir*		Specifies a preprocessor file directory. By default, the compiler uses the current directory.	Section 2.3.10
--temp_directory=*directory*	-ft	Specifies a temporary file directory. By default, the compiler uses the current directory.	Section 2.3.10

Table 2-18. Default File Extensions Options

Option	Alias	Effect	Section
--asm_extension=[.]*extension*	-ea	Sets a default extension for assembly source files.	Section 2.3.9
--c_extension=[.]*extension*	-ec	Sets a default extension for C source files.	Section 2.3.9
--cpp_extension=[.]*extension*	-ep	Sets a default extension for C++ source files.	Section 2.3.9
--listing_extension=[.]*extension*	-es	Sets a default extension for listing files.	Section 2.3.9
--obj_extension=[.]*extension*	-eo	Sets a default extension for object files.	Section 2.3.9

Table 2-19. 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 2.3.2

2.3.1 Linker Options

The following tables list the linker options. See Chapter 4 of this document and the *TMS320C28x Assembly Language Tools User's Guide* for details on these options.

Table 2-20. 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* words and defines a global symbol that specifies the stack size. Default = 1K words.
--heap_size=*size*	[-]-heap	Sets heap size (for the dynamic memory allocation in C) to *size* words and defines a global symbol that specifies the heap size. Default = 1K words.
--warn_sections	-w	Displays a message when an undefined output section is created.

Table 2-21. 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 4.3.1.1.
--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 2-22. 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 2-23. 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.
--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.

Table 2-24. 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.
--run_abs	-abs	Produces an absolute listing file.
--xml_link_info=*file*		Generates a well-formed XML *file* containing detailed information about the result of a link.

Table 2-25. 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_sym_merge	-b	Disables merge of symbolic debugging information in COFF object files.
--no_symtable	-s	Strips symbol table information and line number entries from the executable object file.
--retain		Retains a list of sections that otherwise would be discarded. (EABI only)
--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. The --symbol_map option is supported when used with --opt_level=4.

Table 2-25. Symbol Management Options (continued)

Option	Alias	Description
--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 2-26. 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. The default if this option is specified with no *type* is lzss for Lempel-Ziv-Storer-Szymanski compression. (EABI only)
--copy_compression[=type]		Compresses data copied by linker copy tables. The default if this option is specified with no *type* is lzss for Lempel-Ziv-Storer-Szymanski compression. (EABI only)
--fill_value=*value*	-f	Sets default fill value for holes within output sections
--ram_model	-cr	Initializes variables at load time. See Section 4.3.5 for details.
--rom_model	-c	Autoinitializes variables at run time. See Section 4.3.5 for details.

Table 2-27. Link-Time Optimization Options

Option(1)	Alias	Description
--keep_asm		Retain any post-link files (.pl) and .absolute listing files (.abs) generated by the −plink option. This allows you to view any changes the post-link optimizer makes. (Requires use of -plink)
--no_postlink_across_calls	-nf	Disable post-link optimizations across functions. (Requires use of -plink)
--plink_advice_only		Annotates assembly code with comments if changes cannot be made safely due to pipeline considerations, such as when float support or VCU support is enabled. (Requires use of -plink)
--postlink_exclude	-ex	Exclude files from post-link pass. (Requires use of -plink)
--postlink_opt	-plink	Post-link optimizations (only after -z).

(1) See Section 5.6 for details.

Table 2-28. Miscellaneous Options

Option	Alias	Description
--compress_dwarf[=off|on]		Aggressively reduces the size of DWARF information from input object files. Default is off.
--disable_clink	-j	Disables conditional linking of COFF object files. (COFF only)
--linker_help	[-]-help	Displays information about syntax and available options.
--preferred_order=*function*		Prioritizes placement of functions.
--strict_compatibility[=off|on]		Performs more conservative and rigorous compatibility checking of input object files. Default is on.
--unused_section_elimination[=off|on]		Eliminates sections that are not needed in the executable module. Default is on. (EABI only)
--zero_init=[off|on]		Controls preinitialization of uninitialized variables. Default is on. Always off if --ram_model is used. (EABI only)

2.3.2 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 3.10. 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:

```
cl2000 --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 C2000_C_OPTION environment variable and you do not want to link. See Section 4.1.3.

--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, the *name* is set to 1. The --define 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 2.5.2.1.

--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. The --keep_asm 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 4.1.

--skip_assembler
Compiles only. The specified source files are compiled but not assembled or linked. The --skip_assembler option's short form is -n. This option overrides --run_linker. The output is assembly language output from the compiler.

--src_interlist
Invokes the interlist feature, which interweaves optimizer comments *or* C/C++ source with assembly source. If the optimizer is invoked (--opt_level=*n* option), optimizer comments are interlisted with the assembly language output of the compiler, which may rearrange code significantly. If the optimizer is not invoked, C/C++ source statements are interlisted with the assembly language output of the compiler, which allows you to inspect the code generated for each C/C++ statement. The --src_interlist option implies the --keep_asm option. The --src_interlist option's short form is -s.

--tool_version
Prints the version number for each tool in the compiler. No compiling occurs.

--undefine=*name*
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.

2.3.3 Miscellaneous Useful Options

Following are detailed descriptions of miscellaneous options:

--advice:performance[=all|none]
When Trigonometric Math Unit (TMU) support is enabled (--tmu_support) and --fp_mode=strict, advice will be generated if the compiler encounters function calls that could be replaced with TMU hardware instructions under --fp_mode=relaxed. These include floating point division, sqrt, sin, cos, atan, and atan2.
In addition, when compiling for EABI with --float_support=fpu32, enabling this option causes advice to be provided if double floating point operations are detected. Since EABI doubles are 64 bits, consider changing doubles to floats for improved performance in FPU32-mode.

--float_operations_allowed= {none|all|32|64}
Restricts the type of floating point operations allowed in the application. The default is all. If set to none, 32, or 64, the application is checked for operations that will be 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:

- Division by a constant is converted to inverse multiplication.
- Certain C standard float functions--such as sqrt, sin, cos, atan, and atan2--are redirected to optimized inline functions where possible.
- If the --tmu_support option is used to enable support for the Trigonometric Math Unit (TMU) and relaxed floating-point mode is enabled, RTS library calls are replaced with the corresponding TMU hardware instructions for the following floating-point operations: floating point division, sqrt, sin, cos, atan, and atan2. Note that there are algorithmic differences between the TMU hardware instructions and the library routines, so the results of operations may differ slightly.
- If --tmu_support=tmu1 is used with --fp_mode=relaxed, special "relaxed" versions of the following 32-bit RTS math functions are used: exp2f(), expf(), log2f(), logf(), and powf(). Note that relaxed versions that work with double types are not provided.

--fp_reassoc={on|off}
Enables or disables the reassociation of floating-point arithmetic. The default is on. 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.
When --fp_reassoc=on, RPT MACF32 instructions may be generated. Because the RPT MACF32 instruction computes two partial sums and adds them together afterward to compute the entire accumulation, the result can vary in precision from a serial floating-point multiply accumulate loop.

--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.

2.3.4 Run-Time Model Options

These options are specific to the TMS320C28x toolset. See the referenced sections for more information. TMS320C28x-specific assembler options are listed in Section 2.3.11.

The C28x compiler supports both the COFF ABI and the Embedded Application Binary Interface (EABI) ABI. EABI uses the ELF object format and the DWARF debug format.

--abi={eabi|coffabi}	Specifies the application binary interface (ABI). The default ABI is COFF. EABI is also supported. See Section 2.13. Refer to the *C28x Embedded Application Binary Interface* (SPRAC71) application report. All code in an EABI application must be built for EABI. Make sure all your libraries are available in EABI mode before migrating your existing COFF ABI systems to EABI.
--cla_support={cla0|cla1|cla2}	Specifies TMS320C28x Control Law Accelerator (CLA) Type 0, Type 1, or Type 2 support. This option is used to compile or assemble code written for the CLA. This option does not require any special library support when linking; the libraries used for C28x with/without FPU support should be sufficient.
--float_support={ fpu32 | fpu64 | softlib }	Specifies use of TMS320C28x 32-bit or 64-bit hardware floating-point support. Using --float_support=fpu32 specifies the C28x architecture with 32-bit hardware floating-point support. Using --float_support=fpu64 specifies the C28x architecture with 64-bit hardware floating-point support. FPU64 is supported only when using EABI. If the --tmu_support option is used to enable support for the Trigonometric Math Unit, the --float_support option is automatically set to fpu32. The default is softlib, which performs floating-point calculations without special hardware support.
--idiv_support={ none | idiv0 }	Enables support for fast integer division using hardware extensions to provide a set of instructions to accelerate integer division. If this hardware is available, use --idiv_support=idiv0 to cause these instructions to be used. The default is none. Datasheets for devices that include this hardware contain the words "Support for Fast Integer Division (FINTDIV)." (EABI only.) When this option is enabled, the built-in integer division and modulo operators ("/" and "%") use the appropriate faster instructions. See Section 7.8.2 for more about such cases. When this option is enabled, you can also use the fast integer division intrinsics described in Section 7.6.4. In order to use these intrinsics, your code must include the `stdlib.h` header file.
--no_rpt	Prevents the compiler from generating repeat (RPT) instructions. By default, repeat instructions are generated for certain memcpy, division, and multiply-accumulate operations. However, repeat instructions are not interruptible.
--pending_instantiations=#	Specify the number of template instantiations that may be in progress at any given time. Use 0 to specify an unlimited number.

--protect_volatile=*num*

Enables volatile reference protection. Pipeline conflicts may occur between non-local variables that have been declared volatile. A conflict can occur between a write to one volatile variable that is followed by a read from a different volatile variable. The --protect_volatile option allows at least *num* instructions to be placed between the two volatile references to ensure the write occurs before the read. The num is optional. If no num is given, the default value is 2. For example, if --protect_volatile=4 is used, volatile writes and volatile reads are protected by at least 4 instructions.

The peripheral pipeline protection hardware protects all internal peripherals and XINTF zone 1. If you connect peripherals to Xintf zone 0, 2, 6, 7 then you may need to use the --protect_volatile option. Hardware protection or using this option is not required for memories.

--ramfunc={on|off}

If set to on, specifies that all functions should be placed in the .TI.ramfunc section, which is placed in RAM. If set to off, only functions with the ramfunc function attribute are treated this way. See Section 6.15.2.

Newer TI linker command files support the --ramfunc option automatically by placing functions in the .TI.ramfunc section. If you have a linker command file that does not include a section specification for the .TI.ramfunc section, you can modify the linker command file to place this section in RAM. See the *TMS320C28x Assembly Language Tools User's Guide* for details on section placement.

--rpt_threshold=*k*

Generates RPT loops that iterate *k* times or less (k is a constant between 0 and 256). Multiple RPT's may be generated for the same loop, if iteration count is more than k and if code size does not increase too much. Using this option when optimizing for code size disables RPT loop generation for loops whose iteration count can be greater than k.

Note that inlined memcpy calls now support >255 words through the use of RPT with a register operand. This allows inlining of memcpy of up to 65535 words. If you set the --no_rpt or --rpt_threshold option, such inlining is disabled or reduced, respectively. The maximum value that may be specified with --rpt_threshold is still 256.

--silicon_errata_fpu1_workaround=on|off

Enabling this option prevents FPU register write conflicts that can occur during certain instructions. CPU-to-FPU register writes cannot occur during FRACF32, F32TOUI32, or UI16TOF32 instructions. If you enable this option, the compiler adds five NOP instructions before these instructions to prevent conflicts.

This option is disabled by default if any of the following options are enabled: --float_support=fpu64, --tmu_support, or --vcu_support=vcu2|vcrc.

--silicon_version=28

Generates code for the TMS320C28x architecture. The only value accepted is 28. This is the default, so this option is no longer required on command lines.

--unified_memory

Use the --unified_memory (-mt) option if your memory map is configured as a single unified space; this allows the compiler to generate RPT PREAD instructions for most memcpy calls and structure assignments. This also allows MAC instructions to be generated. The --unified_memory option also allows more efficient data memory instructions to be used to access switch tables.

Even with unified memory, memory for some peripherals and some RAM associated with those peripherals is allocated only in data memory. If –unified_memory is enabled, you can prevent program memory address access to specific symbols by declaring symbols as volatile.

--tmu_support[=tmu0|tmu1]

Enables support for the Trigonometric Math Unit (TMU). Using this option automatically enables FPU32 support (as with the --float_support=fpu32 option). When TMU support is enabled, intrinsics are available to perform trigonometric instructions on the TMU.

There are algorithmic differences between the TMU hardware instructions and the library routines, so the results of operations may differ slightly.

The tmu1 setting is available with EABI only. The tmu1 setting adds support for the LOG2F32 and IEXP2F32 intrinsics in addition to the intrinsics supported with the tmu0 setting.

In relaxed floating point mode, RTS library calls are replaced with the corresponding TMU hardware instructions for the following floating-point operations: floating point division, sqrt, sin, cos, atan, and atan2. Additionally, if the --tmu_support=tmu1 option is used with --fp_mode=relaxed, special versions of the following 32-bit float math functions are used: exp2f(), expf(), log2f(), logf(), and powf(). Relaxed versions that work with EABI's 64-bit double types are not provided.

--vcu_support[=vcu0|vcu2|vcrc]

The vcu0 and vcu2 settings specify there is support for Type 0 or Type 2 of the Viterbi, Complex Math and CRC Unit (VCU). Note that there is no VCU Type 1. The default is vcu0. The vcrc setting specifies support for Cyclic Redundancy Check (CRC) algorithms only. Support for vcrc is available only if FPU32 or FPU64 is used.
This option is useful only if the source is in assembly code, written for the VCU. The option is ignored for C/C++ code. This option does not need any special library support when linking; the libraries used for C28x with/without VCU support should be sufficient. Also note that there is no VCU Type 1.

2.3.5 Symbolic Debugging and Profiling Options

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

--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 3.12. For details on the DWARF format, see *The DWARF Debugging Standard*.

--symdebug:dwarf_version={2|3|4}

Specifies the DWARF debugging format version (2, 3, or 4) to be generated when --symdebug:dwarf (the default) is specified. By default, the compiler generates DWARF version 3 debug information for the COFF ABI and version 4 for EABI. DWARF versions 2, 3, and 4 may be intermixed safely. When DWARF 4 is used, type information is placed in the .debug_types section. At link time, duplicate type information is removed. This method of type merging is superior to DWARF 2 or 3 and results in a smaller executable. In addition, DWARF 4 reduces the size of intermediate object files in comparison to DWARF 3. For more about TI extensions to the DWARF language, see *The Impact of DWARF on TI Object Files* (SPRAAB5).

--symdebug:none

Disables all symbolic debugging output. This option is not recommended; it prevents debugging and most performance analysis capabilities.

--symdebug:profile_coff

Adds the necessary debug directives to the object file which are needed by the profiler to allow function level profiling with minimal impact on optimization (when used). This option does not hinder optimization.
You can set breakpoints and profile on function-level boundaries in Code Composer Studio, but you cannot single-step through code as with full debug ability. (COFF only; not supported for EABI.)

--symdebug:skeletal

Deprecated. Has no effect.

2.3.6 Specifying Filenames

The input files that you specify on the command line can be C source files, C++ source files, assembly source 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

Note

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

For information about how you can alter the way that the compiler interprets individual filenames, see Section 2.3.7. For information about how you can alter the way that the compiler interprets and names the extensions of assembly source and object files, see Section 2.3.10.

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:

```
cl2000 *.cpp
```

Note

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

2.3.7 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:

--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:

```
cl2000 --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.

2.3.8 Changing How the Compiler Processes C Files

The --cpp_default option causes the compiler to process C files as C++ files. By default, the compiler treats files with a .c extension as C files. See Section 2.3.9 for more information about filename extension conventions.

2.3.9 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:

--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:

```
cl2000 --asm_extension=.rrr --obj_extension=.o fit.rrr
```

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

```
cl2000 --asm_extension=rrr --obj_extension=o fit.rrr
```

2.3.10 Specifying Directories

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

--abs_directory=*directory*
Specifies the destination directory for absolute listing files. The default is to use the same directory as the object file directory. For example:
```
cl2000 --abs_directory=d:\abso_list
```

--asm_directory=*directory*
Specifies a directory for assembly files. For example:
```
cl2000 --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:
```
cl2000 --list_directory=d:\listing
```

--obj_directory=*directory*
Specifies a directory for object files. For example:
```
cl2000 --obj_directory=d:\object
```

--output_file=*filename*
Specifies a compilation output file name; can override --obj_directory. For example:
```
cl2000 --output_file=transfer
```

--pp_directory=*directory*
Specifies a preprocessor file directory for object files (default is .). For example:
```
cl2000 --pp_directory=d:\preproc
```

--temp_directory=*directory*
Specifies a directory for temporary intermediate files. For example:
```
cl2000 --temp_directory=d:\temp
```

2.3.11 Assembler Options

Following are assembler options that you can use with the compiler. For more information, see the *TMS320C28x Assembly Language Tools User's Guide.*

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

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

--issue_remarks Issues remarks (non-serious warnings). For the assembler, enables additional assembly-time checking. A remark is generated if an .ebss allocation size is greater than 64 words, or if a 16-bit immediate operand value resides outside of the -32 768 to 65 535 range.

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

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

--flash_prefetch_warn Enables assembler warnings if a program data access instruction follows within 8 words of a BF or SBF instruction. As outlined in the *TMS320C281X/TMS320F281X DSP Silicon Errata* (SPRZ193) advisory on the "Flash and OTP Prefetch Buffer Overflow", the flash prefetch buffer may overflow if this instruction sequence is executed from flash or One-Time-Programmable (OTP) memory with the flash prefetch buffer enabled. Whether or not an overflow actually occurs depends on the instruction sequence, flash wait states, and CPU pipeline stalls. If an overflow occurs, it will result in execution of invalid opcodes. Instructions that use program memory addressing include MAC/XMAC, DMAC/XMACD, QMACL, IMACL, PREAD/XPREAD, and PWRITE/XPWRITE.

--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.

--preproc_asm Expands macros in an assembly file and assembles the expanded file. Expanding macros helps you to debug the assembly file. The --preproc_asm option affects only the assembly file. When --preproc_asm is used, the compiler first invokes the assembler to generate the macro-expanded source .exp file. Then the .exp file is assembled to generate the object file. The debugger uses the .exp file for debugging. The .exp file is an intermediate file and any update to this file will be lost. You need to make any updates to the original assembly file.

2.3.12 Deprecated Options

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.

2.4 Controlling the Compiler Through Environment Variables

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

Note

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

2.4.1 Setting Default Compiler Options (C2000_C_OPTION)

You might find it useful to set the compiler, assembler, and linker default options using the C2000_C_OPTION environment variable. If you do this, the compiler uses the default options and/or input filenames that you name C2000_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 C2000_C_OPTION environment variable and processes it.

The table below shows how to set the C2000_C_OPTION environment variable. Select the command for your operating system:

Operating System	Enter
UNIX (Bourne shell)	**C2000_C_OPTION="** *option*$_1$ [*option*$_2$ . . .]**"; export C2000_C_OPTION**
Windows	**set C2000_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 C2000_C_OPTION environment variable as follows:

```
set C2000_C_OPTION=--quiet --src_interlist --run_linker
```

Any options following --run_linker on the command line or in C2000_C_OPTION are passed to the linker. Thus, you can use the C2000_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 C2000_C_OPTION is set as shown above:

```
cl2000 *c                        ; compiles and links
cl2000 --compile_only *.c        ; only compiles
cl2000 *.c --run_linker lnk.cmd  ; compiles and links using a command file
cl2000 --compile_only *.c --run_linker lnk.cmd
       ; only compiles (--compile_only overrides --run_linker)
```

For details on compiler options, see Section 2.3. For details on linker options, see the *Linker Description* chapter in the *TMS320C28x Assembly Language Tools User's Guide*.

2.4.2 Naming One or More Alternate Directories (C2000_C_DIR)

The linker uses the C2000_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)	**C2000_C_DIR="** *pathname*$_1$ **;** *pathname*$_2$ **;..."; export C2000_C_DIR**
Windows	**set C2000_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 C2000_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 C2000_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 C2000_C_DIR`
Windows	`set C2000_C_DIR=`

2.5 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.

2.5.1 Predefined Macro Names

The compiler maintains and recognizes the predefined macro names listed in Table 2-29.

Table 2-29. Predefined C28x Macro Names

Macro Name	Description
__DATE__ (1)	Expands to the compilation date in the form *mmm dd yyyy*
__FILE__ (1)	Expands to the current source filename
__LINE__ (1)	Expands to the current line number
__little_endian__	Always defined to 1.
__PTRDIFF_T_TYPE__	Set to the type of ptrdiff_t.
__SIZE_T_TYPE__	Set to the type of size_t.
__STDC__ (1)	Defined to 1 to indicate that compiler conforms to ISO C Standard. See Section 6.1 for exceptions to ISO C conformance.
__STDC_VERSION__	C standard macro.
__STDC_HOSTED__	C standard macro. Always defined to 1.

Table 2-29. Predefined C28x Macro Names (continued)

Macro Name	Description
__STDC_NO_THREADS__	C standard macro. Always defined to 1.
__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__	Defined to 1 if --abi=eabi is used.
__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.
__TIME__ (1)	Expands to the compilation time in the form "*hh:mm:ss*"
__TMS320C2000__	Defined for C28x processor
__TMS320C28XX__	Defined if target is C28x
__TMS320C28XX_CLA__	Defined to 1 if any --cla_support option was used and the source file is a .cla file.
__TMS320C28XX_CLA0__	Defined to 1 if the --cla_support=cla0 option was used and the source file is a .cla file.
__TMS320C28XX_CLA1__	Defined to 1 if the --cla_support=cla1 option was used and the source file is a .cla file.
__TMS320C28XX_CLA2__	Defined to 1 if the --cla_support=cla2 option was used and the source file is a .cla file.
__TMS320C28XX_FPU32__	Defined to 1 if either the --float_support=fpu32 or fpu64 option was used.
__TMS320C28XX_FPU64__	Defined to 1 if the --float_support=fpu64 option was used.
__TMS320C28XX_IDIV__	Defined to 1 if the --idiv_support=idiv0 option was used.
__TMS320C28XX_TMU__	Defined to 1 if the --tmu_support option was used with any setting.
__TMS320C28XX_TMU0__	Defined to 1 if the --tmu_support option was used with any setting.
__TMS320C28XX_TMU1__	Defined to 1 if the --tmu_support=tmu1 option was used.
__TMS320C28XX_VCU0__	Defined to 1 if the --vcu_support option was used with any setting.
__TMS320C28XX_VCU2__	Defined to 1 if the --vcu_support=vcu2 option was used.
__TMS320C28XX_VCRC__	Defined to 1 if the --vcu_support=vcrc option was used.
__WCHAR_T_TYPE__	Set to the type of wchar_t.
_INLINE	Expands to 1 if optimization is used (--opt_level or -O option); undefined otherwise.

(1) Specified by the ISO standard

You can use the names listed in Table 2-29 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");
```

2.5.2 The Search Path for #include Files

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

- If you enclose the filename in double quotes (" "), the compiler searches for the file in 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 C2000_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 C2000_C_DIR environment variable.

See Section 2.5.2.1 for information on using the --include_path option. See Section 2.4.2 for more information on input file directories.

2.5.2.1 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	`cl2000 --include_path=/tools/files source.c`
Windows	`cl2000 --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 C2000_C_DIR environment variable.

For example, if you set up C2000_C_DIR with the following command:

```
C2000_C_DIR "/usr/include;/usr/ucb"; export C2000_C_DIR
```

or invoke the compiler with the following command:

```
cl2000 --include_path=/usr/include file.c
```

and file.c contains this line:

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

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

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

2.5.3 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.

2.5.4 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.

2.5.5 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.

2.5.6 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.

2.5.7 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.

2.5.8 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.

2.5.9 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.

2.5.10 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.

2.6 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 the contents of 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.

2.7 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.

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.

2.7.1 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 2.7.
--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).
--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.

2.7.2 How You Can Use Diagnostic Suppression Options

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

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

If you invoke the compiler with the --quiet option, this is the result:

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

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

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

Next, you can use the diagnostic identifier of 111 as the argument to the --diag_remark option to treat this warning as a remark. This compilation 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.

2.8 Other Messages

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

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

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

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

sym-id name X filename line number column number

sym-id	An integer uniquely assigned to each identifier	
name	The identifier name	
X	One of the following values:	
	D	Definition
	d	Declaration (not a definition)
	M	Modification
	A	Address taken
	U	Used
	C	Changed (used and modified in a single operation)
	R	Any other kind of reference
	E	Error; reference is indeterminate
filename	The source file	
line number	The line number in the source file	
column number	The column number in the source file	

2.10 Generating a Raw Listing File (--gen_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 2-30.

Table 2-30. 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 2-31.

Table 2-31. 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 2-31 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 2.7.

2.11 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 2.11.2). 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 may be inlined at link time if the file containing the definition and the call site were both compiled with --opt_level=4. Functions defined as both static and inline are more likely to be inlined.
- When --opt_level=3 or greater 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 3.5.
- The pragma FUNC_ALWAYS_INLINE (Section 6.9.11) and the equivalent `always_inline` attribute (Section 6.15.2) 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 6.9.9) 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.

2.11.1 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 generally inlined whether or not you use the optimizer. However, if the --opt_for_speed option is set to level 0 or 1, the compiler may choose not to inline intrinsics that expand to a substantial number of instructions. For example, the integer division intrinsics enabled by the --idiv_support=idiv0 option expand to more instructions than most intrinsics.

For details about intrinsics, and a list of the intrinsics, see Section 7.6. In addition to those listed, abs and memcpy are implemented as intrinsics.

2.11.2 Inlining Restrictions

The compiler makes decisions about which functions to inline based on the factors mentioned in Section 2.11. 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 and -O4 optimization is not used

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, then the one that appears first will be used and the rest will be ignored.

2.11.3 Unguarded Definition-Controlled Inlining

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

You must invoke the optimizer with any --opt_level option to turn on definition-controlled inlining. Automatic inlining is also turned on when using --opt_level=3.

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

Example 2-1. Using the Inline Keyword

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

2.11.4 Guarded Inlining and the _INLINE Preprocessor Symbol

When declaring a function in a header file as static inline, you must follow additional procedures to avoid a potential code size increase the optimizer is not run.

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

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

In the following examples there are two definitions of the strlen function. The first (Example 2-2), in the header file, is an inline definition. This definition is enabled and the prototype is declared as static inline only if _INLINE is true (_INLINE is automatically defined for you when the optimizer is used).

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

Example 2-2. Header File string.h

```
/*****************************************************************************/
/* string.h  vx.xx                                                           */
/* Copyright (c) 1993-2006 Texas Instruments Incorporated                    */
/* Excerpted ...                                                             */
/*****************************************************************************/
#ifdef _INLINE
#define _IDECL static inline
#else
#define _IDECL extern _CODE_ACCESS
#endif
_IDECL size_t  strlen(const char *_string);
#ifdef _INLINE
/*****************************************************************************/
/*  strlen                                                                   */
/*****************************************************************************/
static inline size_t strlen(const char *string)
{
   size_t      n = (size_t)-1;
   const char *s = string - 1;
   do n++; while (*++s);
   return n
}
#endif
```

Example 2-3. Library Definition File

```
/*****************************************************************************/
/*  strlen                                                                   */
/*****************************************************************************/
#undef _INLINE
#include <string>
_CODE_ACCESS size_t strlen(cont char * string)
{
   size_t      n = (size_t)-1;
   const char *s = string - 1;
   do n++; while (*++s);
   return n;
}
```

2.12 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:

```
cl2000 --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 3.10. Using the --c_src_interlist option can cause performance and/or code size degradation.

The following example shows a typical interlisted assembly file.

```
;--------------------------------------------------------------------------
; 1 | int main()
;--------------------------------------------------------------------------
;*************************************************************
;* FNAME: _main                         FR SIZE: 0           *
;*                                                           *
;* FUNCTION ENVIRONMENT                                      *
;*                                                           *
;* FUNCTION PROPERTIES                                       *
;*                          0 Parameter, 0 Auto, 0 SOE       *
;*************************************************************
_main:
;--------------------------------------------------------------------------
; 3 | printf("Hello World\n");
;--------------------------------------------------------------------------
        MOVL    XAR4,#SL1               ; |3|
        LCR     #_printf                ; |3|
        ; call occurs [#_printf] ; |3|
;--------------------------------------------------------------------------
; 4 | return 0;
;--------------------------------------------------------------------------
;*************************************************************
;* STRINGS *
;*************************************************************
        .sect   ".econst"
SL1:    .string "Hello World",10,0
;*************************************************************
;* UNDEFINED EXTERNAL REFERENCES *
;*************************************************************
.global _printf
```

2.13 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 source, and allows the resulting executable to run on any system that supports that ABI.

Object files conforming to different ABIs cannot be linked together. The linker detects this situation and generates an error.

The C28x compiler supports two ABIs. The ABI is chosen through the --abi option as follows:

- **COFF ABI** (--abi=coffabi)

  The COFF ABI is the original ABI format. It is the default.
- **EABI** (--abi=eabi)

  Use this option to select the C28x Embedded Application Binary Interface (EABI).

All code in an application must be built for the same ABI. Make sure all your libraries are available in EABI mode before migrating COFF ABI applications to EABI.

For more details on the ABI, see Section 6.11.

2.14 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 6.9.20 for information about the NO_HOOKS pragma.

2.15 Live Firmware Update (LFU)

Note

This functionality is supported only when used with EABI. It is not supported with the COFF ABI.

High-availability systems may need to be designed so that the firmware can be upgraded without taking the system offline. Examples include systems that service data centers, hospitals, and military applications. The ability to update system firmware while the system is running and start using the new firmware once the update is complete is called Live Firmware Update (LFU). This is also described as a "warm" start.

For information about creating and calling a custom entry point that performs the warm start, see the *Live Firmware Update Reference Design with C2000 MCUs* (TIDUEY4) design guide.

In order to support LFU, code generation tools need to provide a way to ensure that no system reset occurs during an update, no real-time interrupts are missed, and the system state (global and static variables) can be maintained. Both program and data memory of the firmware image may need to be updated. Global and static variables may be handled in the following ways during a warm start:

- **Preserve:** The values stored in such symbols are retained from before the warm start. The addresses of such symbols are unchanged from the address in the reference ELF image. Each such symbol has a `.TI.bound` section. If the .TI.bound sections are contiguous in memory, the linker can coalesce them into a single output section, which reduces the number of CINIT records required to initialize them. (This is the default if neither --lfu_default nor an attribute is used to specify otherwise.)
- **Update:** The values stored for such symbols are re-initialized during a warm start. The addresses of such symbols may change compared to the addresses in the reference ELF image. Such symbols are collected by the linker into a single `.TI.update` output section. This section defaults to copy compression (that is, no decompression is required during a warm start), which reduces the LFU image switchover time.
- **Allow move:** These symbols can be allocated at any memory address during a warm start. The variables are not re-initialized, so their values are unspecified. This behavior occurs only if the `--lfu_default=none` option is used and a global or static variable has neither the "preserve" nor "update" attribute.

The C28x and CLA compilers provide LFU support for ELF-based firmware images, which use the EABI application binary interface. This support allows you to switch over to a new LFU image while choosing whether to preserve, update (re-initialize), or add individual global and static symbols read in from a reference ELF binary.

Features provided to support LFU functionality include:

- The --lfu_reference_elf compiler option, which points to the previous ELF executable binary to use as a reference to obtain a list of the memory addresses of global and static symbols. See Section 2.3.
- The --lfu_default compiler option, which may be used to set the default for how global and static symbols are handled during a warm start. See Section 2.3.
- The `preserve` attribute, which can be used in C code to specify that the address of an individual symbol should be preserved. See Section 6.15.4.
- The `update` attribute, which can be used in C code to specify that an individual symbol should be re-initialized during a warm start. See Section 6.15.4.
- The .TI.update and .TI.bound sections, which the linker uses to collect symbols that need to be updated and preserved (respectively) during a warm start. See Section 7.1.1.
- The __TI_auto_init_warm() RTS routine, which should be called from a custom entry point during warm start. See Section 8.4.
- The .preserve assembler directive, which can be used in assembly code to specify that the address of an individual symbol should be preserved. To specify that a symbol should be re-initialized during a warm start, no special directive is required. See the "Assembler Directives" chapter of the *TMS320C28x Assembly Language Tools User's Guide* (SPRU513).

Chapter 3

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, rearranging statements and expressions, and allocating variables into registers.

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

3.1 Invoking Optimization

The C/C++ compiler is able to perform various optimizations. High-level optimizations are performed in the optimizer and low-level, target-specific optimizations occur in the code generator. Use higher optimization levels, such as --opt_level=2 and --opt_level=3, to achieve optimal code.

The easiest way to invoke optimization is to use the compiler program, specifying the --opt_level=*n* option on the compiler command line. You can use -O*n* to alias the --opt_level option. The *n* denotes the level of optimization (0, 1, 2, 3, and 4), 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
  - Allocates variables to registers
  - Performs loop rotation
  - Eliminates unused code
  - Simplifies expressions and statements
  - Expands calls to functions declared inline
- **--opt_level=1** or **-O1**

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

  Performs all --opt_level=1 (-O1) optimizations, plus:
  - Performs loop optimizations
  - Eliminates global common subexpressions
  - Eliminates global unused assignments
  - Performs loop unrolling
- **--opt_level=3** or **-O3**

  Performs all --opt_level=2 (-O2) optimizations, plus:
  - Removes all functions that are never called
  - Simplifies functions with return values that are never used
  - Inlines calls to small functions
  - Reorders function declarations; the called functions attributes are known when the caller is optimized
  - Propagates arguments into function bodies when all calls pass the same value in the same argument position
  - Identifies file-level variable characteristics

  If you use --opt_level=3 (-O3), see Section 3.3 and Section 3.4 for more information.
- **--opt_level=4** or **-O4**

  Performs link-time optimization. See Section 3.6 for details.

For details about how the --opt_level and --opt_for_speed options and various pragmas affect inlining, see Section 2.11.

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

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

3.2 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 2

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

A fast branch (BF) instruction is generated by default when the --opt_for_speed option is used. When --opt_for_speed is not used, the compiler generates a BF instruction only when the condition code is one of NEQ, EQ, NTC and TC. The reason is that BF with these condition codes can be optimized to SBF. There is a code size penalty to use BF instruction when the condition code is NOT one of NEQ, EQ, NTC and TC. (Fast branch instructions are also generated for functions with the ramfunc function attribute.)

The --no_fast_branch option is deprecated and has no effect.

3.3 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 3-1 work with --opt_level=3 to perform the indicated optimization:

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

If You ...	Use this Option	See
Want to create an optimization information file	--gen_opt_level=*n*	Section 3.3.1
Want to compile multiple source files	--program_level_compile	Section 3.4

3.3.1 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 3-2 to select the appropriate level to append to the option.

Table 3-2. 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

3.4 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). (If you use --opt_level=4 (-O4), the --program_level_compile option cannot be used, because link-time optimization provides the same optimization opportunities as program level optimization.)

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 or higher 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 3.3.1 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.

3.4.1 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 3-3 to select the appropriate level to append to the --call_assumptions option.

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

Table 3-4. 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 3.4.2 for information about these situations.

3.4.2 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 6.9.13) before any declaration or reference to a function that you want to keep.

Another approach you can take when you use assembly functions in your program is to use the --call_assumptions=*n* option with the --program_level_compile and --opt_level=3 options. See Section 3.4.1 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.

3.5 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. Any function larger than the *size* threshold is not automatically inlined. You can use the --auto_inline=*size* option in the following ways:

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

The compiler inlines the function only if the result is less than the size parameter. The compiler measures the size of a function in arbitrary units; however, the optimizer information file (created with the --gen_opt_level=1 or --gen_opt_level=2 option) reports the size of each function in the same units that the --auto_inline option uses.

The --auto_inline=*size* option controls only the inlining of functions that are not explicitly declared as inline. If you do not use the --auto_inline=*size* option, the compiler inlines very small functions.

For information about interactions between command-line options, pragmas, and keywords that affect inlining, see Section 2.11.

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.

3.6 Link-Time Optimization (--opt_level=4 Option)

Link-time optimization is an optimization mode that allows the compiler to have visibility of the entire program. The optimization occurs at link-time instead of compile-time like other optimization levels.

Link-time optimization is invoked using the --opt_level=4 option. This option must be placed *before* the --run_linker (-z) option on the command line, because both the compiler and linker are involved in link-time optimization. At compile time, the compiler embeds an intermediate representation of the file being compiled into the resulting object file. At link-time this representation is extracted from every object file which contains it, and is used to optimize the entire program.

If you use --opt_level=4 (-O4), the --program_level_compile option cannot also be used, because link-time optimization provides the same optimization opportunities as program level optimization (Section 3.4). Link-time optimization provides the following benefits:

- Each source file can be compiled separately. One issue with program-level compilation is that it requires all source files to be passed to the compiler at one time. This often requires significant modification of a customer's build process. With link-time optimization, all files can be compiled separately.
- References to C/C++ symbols from assembly are handled automatically. When doing program-level compilation, the compiler has no knowledge of whether a symbol is referenced externally. When performing link-time optimization during a final link, the linker can determine which symbols are referenced externally and prevent eliminating them during optimization.
- Third party object files can participate in optimization. If a third party vendor provides object files that were compiled with the --opt_level=4 option, those files participate in optimization along with user-generated files. This includes object files supplied as part of the TI run-time support. Object files that were not compiled with –opt_level=4 can still be used in a link that is performing link-time optimization. Those files that were not compiled with –opt_level=4 do not participate in the optimization.
- Source files can be compiled with different option sets. With program-level compilation, all source files must be compiled with the same option set. With link-time optimization files can be compiled with different options. If the compiler determines that two options are incompatible, it issues an error.

3.6.1 Option Handling

When performing link-time optimization, source files can be compiled with different options. When possible, the options that were used during compilation are used during link-time optimization. For options which apply at the program level, --auto_inline for instance, the options used to compile the main function are used. If main is not included in link-time optimization, the option set used for the first object file specified on the command line is used. Some options, --opt_for_speed for instance, can affect a wide range of optimizations. For these options, the program-level behavior is derived from main, and the local optimizations are obtained from the original option set.

Some options are incompatible when performing link-time optimization. These are usually options which conflict on the command line as well, but can also be options that cannot be handled during link-time optimization.

3.6.2 Incompatible Types

During a normal link, the linker does not check to make sure that each symbol was declared with the same type in different files. This is not necessary during a normal link. When performing link-time optimization, however, the linker must ensure that all symbols are declared with compatible types in different source files. If a symbol is found which has incompatible types, an error is issued. The rules for compatible types are derived from the C and C++ standards.

3.7 Using Feedback Directed Optimization

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

3.7.1 Feedback Directed Optimization

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

3.7.1.1 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 pdd2000. The pdd2000 tool consolidates multiple data sets and formats the data into a feedback file (PRF file, see Section 3.7.2) for consumption by phase 2 of feedback directed optimization.

3.7.1.2 Phase 2 -- Use Application Profile Information for Optimization

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

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

3.7.1.3 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 pdd2000 to correlate compiled functions and their associated profile data.

--use_profile_info specifies the profile information file(s) to use for performing phase 2 of feedback directed optimization. More than one profile information file can be specified on the command line; the compiler uses all input data from multiple information files. The syntax for the option is:
--use_profile_info=*file1*[, *file2*, ..., *filen*]
If no filename is specified, the compiler looks for a file named pprofout.prf in the directory where the compiler in invoked.

3.7.1.4 Example Use of Feedback Directed Optimization

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

1. Generate profile information.

```
cl2000 --opt_level=2 --gen_profile_info foo.c --run_linker --output_file=foo.out
     --library=lnk.cmd --library=rts2800_ml.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 pdd2000 on the PDAT files to create a profile information (PRF) file to be used with --use_profile_info.

```
pdd2000 -e foo.out -o pprofout.prf pprofout.pdat
```

4. Re-compile using the profile feedback file.

```
cl2000 --opt_level=2 --use_profile_info=pprofout.prf foo.c --run_linker
     --output_file=foo.out --library=lnk.cmd --library=rts2800_ml.lib
```

3.7.1.5 The .ppdata Section

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

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

3.7.1.6 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.

3.7.1.7 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 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.

3.7.1.8 Invalid Profile Data

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

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

3.7.2 Profile Data Decoder

The code generation tools include a tool called the Profile Data Decoder or pdd2000, which is used for post processing profile data (PDAT) files. The pdd2000 tool generates a profile feedback (PRF) file. See Section 3.7.1 for a discussion of where pdd2000 fits in the profiling flow. The pdd2000 tool is invoked with this syntax:

pdd2000 -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 pdd2000 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

Your application must be compiled with at least DWARF debug support to enable feedback directed optimization. When compiling for feedback directed optimization, the pdd2000 tool relies on basic debug information about each function in generating the formatted .prf file.

The pprofout.pdat file generated by the run-time support is a raw data file of a fixed format understood only by pdd2000. You should not modify this file in any way.

3.7.3 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.

3.7.4 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. You must 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

3.8 Using Profile Information to Analyze Code Coverage

You can use the analysis information from the Profile Data Decoder to analyze code coverage.

3.8.1 Code Coverage

The information collected during feedback directed optimization can be used for generating code coverage reports. As with feedback directed optimization, the program must be compiled with the --gen_profile_info option. Code coverage conveys the execution count of each line of source code in the file being compiled, using data collected during profiling.

3.8.1.1 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 pdd2000. The pdd2000 tool consolidates multiple data sets and formats the data into a feedback file (PRF file, see Section 3.7.2) for consumption by phase 2 of feedback directed optimization.

3.8.1.2 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:

```
cl2000 --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/c2000/deflate.c","_deflateInit2_",216,5,1,"( strm->zalloc )"
```

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

3.8.2 Related Features and Capabilities

The code generation tools provide some features and capabilities that can be used in conjunction with code coverage analysis. The following is a summary:

3.8.2.1 Path Profiler

The code generation tools include a path profiling utility, pprof2000, that is run from the compiler, cl2000. The pprof2000 utility is invoked by the compiler when the --gen_profile or the --use_profile command is used from the compiler command line:

cl2000 --gen_profile ... *file.c*
cl2000 --use_profile ... *file.c*

For further information about profile-based optimization and a more detailed description of the profiling infrastructure, see Section 3.7.

3.8.2.2 Analysis Options

The path profiling utility, pprof2000, appends code coverage 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 . Attempts to mix code coverage and other analysis information in the same output CSV file will be detected, and pprof2000 will emit a fatal error and abort.

--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.

3.8.2.3 Environment Variables

To assist with the management of output CSV analysis files, pprof2000 supports this environment variable:

TI_ANALYSIS_DIR	Specifies the directory in which the output analysis file will be generated.

3.9 Special Considerations When Using Optimization

The compiler is designed to improve your ANSI/ISO-conforming C and C++ programs while maintaining their correctness. However, when you write code for optimization, you should note the special considerations discussed in the following sections to ensure that your program performs as you intend.

3.9.1 Use Caution With asm Statements in Optimized Code

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

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

3.9.2 Use the Volatile Keyword for Necessary Memory Accesses

The compiler analyzes data flow to avoid memory accesses whenever possible. If you have code that depends on memory accesses exactly as written in the C/C++ code, you must use the volatile keyword to identify these accesses. The compiler does not optimize out any references to volatile variables.

In the following example, the loop waits for a location to be read as 0xFF:

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

In this example, *ctrl is a loop-invariant expression, so the loop is optimized down to a single memory read. To correct this, declare ctrl as:

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

3.9.2.1 Use Caution When Accessing Aliased Variables

Aliasing occurs when a single object can be accessed 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 still preserving the correctness of the program. The compiler behaves conservatively.

The compiler assumes that if the address of a local variable is passed to a function, the function might change the local by writing through the pointer, but that it will not make its address available for use elsewhere after returning. For example, the called function cannot assign the local's address to a global variable or return it. In cases where this assumption is invalid, use the -ma compiler option to force the compiler to assume worst-case aliasing. In worst-case aliasing, any indirect reference (that is, using a pointer) can refer to such a variable.

3.9.2.2 Use the --aliased_variables Option to Indicate That the Following Technique Is Used

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

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

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

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

    *p        = 5;   /* p aliases x        */
    *glob_ptr = 10;  /* glob_ptr aliases x */
    h(x);
}

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

3.9.2.3 On FPU Targets Only: Use restrict Keyword to Indicate That Pointers Are Not Aliased

On FPU targets, with --opt_level=2, the optimizer performs dependency analysis. 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 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. This can improve performance and code size, as more FPU operations can be parallelized.

As shown in Example 3-1 and Example 3-2 you can use the restrict keyword to tell the compiler that a and b never point to the same object in foo. Furthermore, the compiler is assured that the objects pointed to by a and b do not overlap in memory.

Example 3-1. Use of the restrict Type Qualifier With Pointers

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

Example 3-2. Use of the restrict Type Qualifier With Pointers

```
void foo(float c[restrict], float d[restrict])
{
   /* foo's code here */
}
```

3.10 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.

```
;****************************************************************
;* FNAME: _copy                          FR SIZE: 0             *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                           0 Parameter, 0 Auto, 0 SOE         *
;****************************************************************
_copy:
;*** 6  ----------------------    if ( n <= 0 ) goto g4;
        CMPB      AL,#0                 ; |6|
        B         L2,LEQ                ; |6|
        ; branch occurs ; |6|
;***    ----------------------    #pragma MUST_ITERATE(1, 4294967295, 1)
:***    ----------------------    L$1 = n-1;
        ADDB      AL,#-1
        MOVZ      AR6,AL
L1:
;***    ----------------------g3:
;*** 7  ----------------------    *str++ = *s++;
;*** 7  ----------------------    if ( (--L$1) != (-1) ) goto g3;
        MOV       AL,*XAR5++            ; |7|
        MOV       *XAR4++,AL            ; |7|
        BANZ      L1,AR6--
        ; branch occurs ; |7|
;***    ----------------------g4:
;***    ----------------------    return;
L2:
        LRETR
        ; return 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.

```
;-----------------------------------------------------------------------
;   2 | int copy (char *str, const char *s, int n)
;-----------------------------------------------------------------------
;***************************************************************
;* FNAME: _copy                         FR SIZE:   0           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;* FUNCTION PROPERTIES                                         *
;*                           0 Parameter,  0 Auto,  0 SOE      *
;***************************************************************
_copy
;* AR4   assigned to _str
;* AR5   assigned to _s
;* AL    assigned to _n
;* AL    assigned to _n
;* AR5   assigned to _s
;* AR4   assigned to _str
;* AR6   assigned to L$1
;*** 6  ----------------------    if ( n <= 0 ) goto g4;
;-----------------------------------------------------------------------
;   4 | int i;
;-----------------------------------------------------------------------
;-----------------------------------------------------------------------
;   6 | for (i = 0; i < n; i++)
;-----------------------------------------------------------------------
        CMPB      AL,#0                 ; |6|
        B         L2,LEQ                ; |6|
        ; branch occurs ; |6|
;***    ----------------------    #pragma MUST_ITERATE(1, 4294967295, 1)
:***    ----------------------    L$1 = n-1;
        ADDB      AL,#-1
        MOVZ      AR6,AL
        NOP
L1:
;*** 7  ----------------------    *str++ = *s++;
;*** 7  ----------------------    if ( (--L$1) != (-1) ) goto g3;
;-----------------------------------------------------------------------
;   7 | *str++ = *s++;
;-----------------------------------------------------------------------
        MOV       AL,*XAR5++            ; |7|
        MOV       *XAR4++,AL            ; |7|
        BANZ      L1,AR6--
        ; branch occurs ; |7|
;***    ----------------------g4:
;***    ----------------------    return;
L2:
        LRETR
        ; return occurs
```

3.11 Data Page (DP) Pointer Load Optimization

When accessing a global variable by name, the compiler uses direct addressing. The C28x supports direct addressing through the data page pointer register, DP. The DP register points to the beginning of a page, which is 64 words long.

To avoid loading the DP for every direct access, the compiler "blocks" some data and sections. Blocking ensures an object is either entirely contained within a single page or is aligned to a 64-word page boundary. Such data page blocking allows the compiler to use the direct addressing mode more often. As a result, it minimizes the need for DP load instructions when accessing global variables known to be stored on a single data page.

For the COFF ABI, all non-const data is blocked.

For EABI, the default blocking rules are:

- Arrays and their sections are not blocked.
- Structs with external linkage (extern in C) are blocked.
- Structs with internal linkage (static in C) are not blocked, but their sections are blocked.

However, data page blocking can result in alignment holes in memory due to aligning data to page boundaries. So, there is a tradeoff between your application's need for code size and speed optimization and its need for data size optimization. You can use the **blocked** and **noblocked** data attributes to control blocking on specific variables. See Section 6.15.4 for details.

Another technique for managing how global variables are organized in memory include grouping global variables that should be stored together in a structure. The DATA_SECTION (Section 6.9.6) and SET_DATA_SECTION (Section 6.9.23) pragmas can also be used to manage data pages.

For examples of C28x data page blocking, see the Data blocking in the C2000 MCU compiler explained topic on the E2E Community.

3.12 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.

3.12.1 Profiling Optimized Code

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

3.13 Increasing Code-Size Optimizations (--opt_for_space Option)

The --opt_for_space option increases the level of code-size optimizations performed by the compiler. These optimizations are done at the expense of performance. The optimizations include procedural abstraction where common blocks of code are replaced with function calls. For example, prolog and epilog code, certain intrinsics, and other common code sequences, can be replaced with calls to functions that are defined in the run-time library. It is necessary to link with the supplied run-time library when using the --opt_for_space option. It is not necessary to use optimization to invoke the --opt_for_space option.

To illustrate how the --opt_for_space option works, the following describes how prolog and epilog code can be replaced. This code is changed to function calls depending on the number of SOE registers, the size of the frame, and whether a frame pointer is used. These functions are defined in each file with the --opt_for_space option, as shown below:

```
_prolog_c28x_1
_prolog_c28x_2
_prolog_c28x_3
_epilog_c28x_1
_epilog_c28x_2
```

Example 3-3 provides an example of C code to be compiled with the --opt_for_space option. The resulting output is shown in Example 3-4.

Example 3-3. C Code to Show Code-Size Optimizations

```
extern int x, y, *ptr;
extern int foo();
int main(int a, int b, int c)
{
   ptr[50] = foo();
   y = ptr[50] + x + y + a +b + c;
}
```

Example 3-4. Example 3-3 Compiled With the --opt_for_space Option

```
FP      .set    XAR2
        .global _prolog_c28x_1
        .global _prolog_c28x_2
        .global _prolog_c28x_3
        .global _epilog_c28x_1
        .global _epilog_c28x_2
        .sect   ".text"
        .global _main
;***************************************************************
;* FNAME: _main                          FR SIZE:   6           *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                           0 Parameter,  0 Auto,  6 SOE       *
;***************************************************************
_main:
        FFC       XAR7,_prolog_c28x_1
        MOVZ      AR3,AR4               ; |5|
        MOVZ      AR2,AH                ; |5|
        MOVZ      AR1,AL                ; |5|
        LCR       #_foo                 ; |6|
        ; call occurs [#_foo] ; |6|
        MOVW      DP,#_ptr
        MOVL      XAR6,@_ptr            ; |6|
        MOVB      XAR0,#50              ; |6|
        MOVW      DP,#_y
        MOV       *+XAR6[AR0],AL        ; |6|
        MOV       AH,@_y                ; |7|
        MOVW      DP,#_x
        ADD       AH,AL                 ; |7|
        ADD       AH,@_x                ; |7|
        ADD       AH,AR3                ; |7|
        ADD       AH,AR1                ; |7|
        ADD       AH,AR2                ; |7|
        MOVB      AL,#0
        MOVW      DP,#_y
        MOV       @_y,AH                ; |7|
        FFC       XAR7,_epilog_c28x_1
        LRETR
        ; return occurs
```

3.14 Compiler Support for Re-Entrant VCU Code

The --isr_save_vcu_regs compiler option generates instructions to save and restore VCU registers using the stack when interrupt service routines occur. This allows VCU code to be re-entrant. If an ISR interrupts a VCU computation, it will not impact results if this option is used.

This option can only be used if the --vcu_support option is set.

- If --vcu_support is set to **vcu0** or **vcu2**, the following instructions are added to the start/end of all ISRs:

```
VMOV32    *SP++,VCRC
VMOV32    *SP++,VSTATUS
    <ISR code here>
VMOV32    VSTATUS,*--SP
VMOV32    VCRC,*--SP
```

- If --vcu_support is set to **vcrc**, the following instructions are added to the start/end of all ISRs:

```
VMOV32    *SP++,VCRC
VMOV32    *SP++,VSTATUS
VMOV32    *SP++,VCRCPOLY
VMOV32    *SP++,VCRCSIZE
    <ISR code here>
VMOV32    VCRCSIZE,*--SP
VMOV32    VCRCPOLY,*--SP
VMOV32    VSTATUS,*--SP
VMOV32    VCRC,*--SP
```

3.15 Compiler Support for Generating DMAC Instructions

The C28x compiler supports DMAC instructions, which perform multiply-accumulate operations on two adjacent signed integers at the same time, optionally shifting the products. A multiply–accumulate operation multiplies two numbers and adds that product to an accumulator. That is, it computes a = a + (b x c). There are three levels of DMAC support that require different levels of C-source modification:

- Generate DMAC instructions automatically from C code (see Section 3.15.1).
- Use assertions for data address alignment to enable DMAC instruction generation (see Section 3.15.2).
- Use the __dmac intrinsic (see Section 3.15.3).

3.15.1 Automatic Generation of DMAC Instructions

The compiler automatically generates DMAC instructions if the compiler recognizes the C-language statements as a DMAC opportunity and the compiler can verify that the data addresses being operated upon are 32-bit aligned. This is the best scenario, because it requires no code modification aside from data alignment pragmas. The following is an example:

```
int src1[N], src2[N];
#pragma DATA_ALIGN(src1,2);          // int arrays must be 32-bit aligned
#pragma DATA_ALIGN(src2,2);

{...}
int i;
long res = 0;
for (i = 0; i < N; i++)              // N must be a known even constant
    res += (long)src1[i] * src2[i];  // Arrays must be accessed via array indices
```

At optimization levels >= -O2, the compiler generates a RPT || DMAC instruction for the example code above if N is a known even constant.

DMAC instructions can also shift the product left by 1 or right by 1 to 6 before accumulation. For example:

```
for (i = 0; i < N; i++)
    res += (long)src1[i] * src2[i] >> 1;   // product shifted right by 1
```

3.15.2 Assertions to Specify Data Address Alignment

In some cases the compiler may recognize a DMAC opportunity in C-language statements but not be able to verify that the data addresses passed to the computation are both 32-bit aligned. In this case, assertions placed in the code can enable the compiler to generate DMAC instructions. The following is an example:

```
int *src1, *src2;    // src1 and src2 are pointers to int arrays of at least size N
                     // You must ensure that both are 32-bit aligned addresses
{...}
int i;
long res = 0;
_nassert((long)src1 % 2 == 0);
_nassert((long)src2 % 2 == 0);
for (i = 0; i < N; i++)             // N must be a known even constant
    res += (long)src1[i] * src2[i]; // src1 and src2 must be accessed via array indices
```

At optimization levels >= -O2, the compiler generates a RPT || DMAC instruction for the example code above if N is a known even constant.

The _nassert intrinsic generates no code and so is not a compiler intrinsic like those listed in Table 7-6. Instead, it tells the optimizer that the expression declared with the assert function is true. This can be used to give hints to the optimizer about what optimizations might be valid. In this example, _nassert is used to assert that the data addresses represented by the src1 and src2 pointers are 32-bit aligned. You are responsible for ensuring that only 32-bit aligned data addresses are passed via these pointers. The code will result in a run-time failure if the asserted conditions are not true.

DMAC instructions can also shift the product left by 1 or right by 1 to 6 before accumulation. For example:

```
for (i = 0; i < N; i++)
    res += (long)src1[i] * src2[i] >> 1;    // product shifted right by 1
```

3.15.3 __dmac Intrinsic

You can force the compiler to generate a DMAC instruction by using the __dmac intrinsic. In this case, you are fully responsible for ensuring that the data addresses are 32-bit aligned.

```
void __dmac(long *src1, long *src2, long &accum1, long &accum2, int shift);
```

- Src1 and src2 must be 32-bit aligned addresses that point to int arrays.
- Accum1 and accum2 are pass-by-reference longs for the two temporary accumulations. These must be added together after the intrinsic to compute the total sum.
- Shift specifies how far to shift the products prior to each accumulation. Valid shift values are -1 for a left shift by 1, 0 for no shift, and 1-6 for a right shift by 1-6, respectively. Note that this argument is required, so you must pass 0 if you want no shift.

See Table 7-6 for details about the __dmac intrinsic.

Example 1:

```
int src1[2N], src2[2N]; // src1 and src2 are int arrays of at least size 2N
                        // You must ensure that both start on 32-bit
                        // aligned boundaries.
{...}
int i;
long res = 0;
long temp = 0;
for (i=0; i < N; i++)             // N does not have to be a known constant
      __dmac(((long *)src1)[i], ((long *)src2)[i], res, temp, 0);
res += temp;
```

Example 2:

```
int *src1, *src2;     // src1 and src2 are pointers to int arrays of at
                      // least size 2N. User must ensure that both are
                      // 32-bit aligned addresses.
{...}
int i;
long res = 0;
long temp = 0;
long *ls1 = (long *)src1;
long *ls2 = (long *)src2;
for (i=0; i < N; i++)     // N does not have to be a known constant
    __dmac(*ls1++, *ls2++, res, temp, 0);
res += temp;
```

In these examples, res holds the final sum of a multiply-accumulate operation on int arrays of length 2*N, with two computations being performed at a time.

Additionally, an optimization level >= -O2 must be used to generate efficient code. Moreover, if there is nothing else in the loop body as in these examples, the compiler generates a RPT || DMAC instruction, further improving performance.

3.16 What Kind of Optimization Is Being Performed?

The TMS320C28x C/C++ compiler uses a variety of optimization techniques to improve the execution speed of your C/C++ programs and to reduce their size. Following are some of the optimizations performed by the compiler:

Optimization	See
Cost-based register allocation	Section 3.16.1
Alias disambiguation	Section 3.16.1
Branch optimizations and control-flow simplification	Section 3.16.3
Data flow optimizations • Copy propagation • Common subexpression elimination • Redundant assignment elimination	Section 3.16.4
Expression simplification	Section 3.16.5
Inline expansion of functions	Section 3.16.6
Function Symbol Aliasing	Section 3.16.7
Induction variable optimizations and strength reduction	Section 3.16.8
Loop-invariant code motion	Section 3.16.9
Loop rotation	Section 3.16.10
Instruction scheduling	Section 3.16.11

C28x-Specific Optimization	See
Register variables	Section 3.16.12
Register tracking/targeting	Section 3.16.13
Tail merging	Section 3.16.14
Autoincrement addressing	Section 3.16.15
Removing comparisons to zero	Section 3.16.16
RPTB generation (for FPU targets only)	Section 3.16.17

3.16.1 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 unroll or eliminate the loop. Strength reduction turns the array references into efficient pointer references with autoincrements.

3.16.2 Alias Disambiguation

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

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

3.16.3 Branch Optimizations and Control-Flow Simplification

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

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

3.16.4 Data Flow Optimizations

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

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

3.16.5 Expression Simplification

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

3.16.6 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 2.11.

3.16.7 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 6.15.2

3.16.8 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.

3.16.9 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.

3.16.10 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.

3.16.11 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 latencies. It can also be used to reduce code size.

3.16.12 Register Variables

The compiler helps maximize the use of registers for storing local variables, parameters, and temporary values. Accessing variables stored in registers is more efficient than accessing variables in memory. Register variables are particularly effective for pointers.

3.16.13 Register Tracking/Targeting

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

3.16.14 Tail Merging

If you are optimizing for code size, tail merging can be very effective for some functions. Tail merging finds basic blocks that end in an identical sequence of instructions and have a common destination. If such a set of blocks is found, the sequence of identical instructions is made into its own block. These instructions are then removed from the set of blocks and replaced with branches to the newly created block. Thus, there is only one copy of the sequence of instructions, rather than one for each block in the set.

3.16.15 Autoincrement Addressing

For pointer expressions of the form *p++, the compiler uses efficient C28x autoincrement addressing modes. In many cases, where code steps through an array in a loop such as below, the loop optimizations convert the array references to indirect references through autoincremented register variable pointers.

```
for (I = 0; I <N; ++I) a(I)...
```

3.16.16 Removing Comparisons to Zero

Because most of the 32-bit instructions and some of the 16-bit instructions can modify the status register when the result of their operation is 0, explicit comparisons to 0 may be unnecessary. The C28x C/C++ compiler removes comparisons to 0 if a previous instruction can be modified to set the status register appropriately.

3.16.17 RPTB Generation (for FPU Targets Only)

When the target has hardware floating-point support, some loops can be converted to hardware loops called repeat blocks (RPTB). Normally, a loop looks like this:

```
Label:
   ...loop body...
   SUB loop_count
   CMP
   B    Label
```

The same loop, when converted to a RPTB loop, looks like this:

```
   RPTB   end_label, loop_count
   ...loop body...
end_label:
```

A repeat block loop is loaded into a hardware buffer and executed for the specified number of iterations. This kind of loop has minimal or zero branching overhead, and can improve performance. The loop count is stored in a special register RB (repeat block register), and the hardware seamlessly decrements the count without any explicit subtractions. Thus, there is no overhead due to the subtract, the compare, and the branch. The only overhead is due to the RPTB instruction that executes once before the loop. The RPTB instruction takes one cycle if the number of iterations is a constant, and 4 cycles otherwise. This overhead is incurred once per loop.

There are limitations on the minimum and maximum loop size for a loop to qualify for becoming a repeat block, due to the presence of the buffer. Also, the loop cannot contain any inner loops or function calls.

This page intentionally left blank.

Chapter 4

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 *TMS320C28x Assembly Language Tools User's Guide*.

4.1 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.

4.1.1 Invoking the Linker Separately

This is the general syntax for linking C/C++ programs as a separate step:

cl2000 --run_linker {**--rom_model | --ram_model**} *filenames*
[*options*] [**--output_file=** *name.out*] **--library=** *library* [*lnk.cmd*]

cl2000 --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 cl2000 --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 4.3.5 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 *TMS320C28x 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 *TMS320C28x 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:

```
cl2000 --run_linker --rom_model prog1 prog2 prog3
     --output_file=prog.out --library=rts2800_ml.lib
```

4.1.2 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:

```
cl2000 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 4.1.1.

All arguments that precede --run_linker on the command line are compiler arguments. These arguments can be C/C++ source files, assembly files, or compiler options. These arguments are described in Section 2.2.

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:

```
cl2000 prog1.c prog2.c prog3.c --run_linker --rom_model --output_file=prog.out
     --library=rts2800_ml.lib
```

When you use cl2000 --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 4.3.5 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 C2000_C_OPTION environment variable

4.1.3 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 C2000_C_OPTION environment variable and want to selectively disable linking with the --compile_only option on the command line.

4.2 Linker Code Optimizations

4.2.1 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.

If this option is not used, the default is "off" for the C28x compiler and "on" for the CLA compiler. If this option is used but neither "on" nor "off" is specified, the default is "on".

4.2.2 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 "off" for the C28x compiler and the CLA compiler. 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.

4.3 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 *TMS320C28x Assembly Language Tools User's Guide*.

4.3.1 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.

4.3.1.1 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 4.3.5 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 4-1. Using the --issue_remarks Option

```
cl2000 --issue_remarks main.c --run_linker --rom_model
<Linking>
remark: linking in "libc.a"
remark: linking in "rts2800_ml.lib" in place of "libc.a"
```

4.3.1.2 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 C2000_C_DIR environment variable for the named library. You can use the --library linker option on the command line or in a command file.

cl2000 --run_linker {**--rom_model** | **--ram_model**} *filenames* **--library=** *libraryname*

4.3.1.3 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.

4.3.2 Run-Time Initialization

You must link all C/C++ programs with code to initialize and execute the program called a bootstrap routine. The bootstrap routine is responsible for the following tasks:

1. Set up status and configuration registers
2. Set up the stack
3. Process the .cinit run-time initialization table to autoinitialize global variables (when using the --rom_model option)
4. Call all global object constructors (.pinit or .init_array depending on the ABI)
5. Call the main() function
6. Call exit() when main() returns

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 4.3.5 for details about using the --rom_model and --ram_model options.

4.3.3 Initialization by the Interrupt Vector

If your C/C++ program begins running at RESET, you must set up the RESET vector to branch to a suitable bootstrap routine, such as _c_int00. You must also make sure the interrupt vectors are included in the project, typically by using the --undef_sym linker option to make a symbol at the start of the interrupt vector a root linker object.

A sample interrupt vector is provided in vectors.c.obj in rts2800_ml.lib. For C28x, the first few lines of the vector are:

```
        .def  _Reset
        .ref  _c_int00
_Reset: .vec _c_int00, USE_RETA
```

4.3.4 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 7.10.3.4 discusses the format of the global constructor table.

4.3.5 Specifying the Type of Global Variable Initialization

The C/C++ compiler produces data tables for initializing global variables. Section 7.10.3.1 discusses the format of these initialization tables. The initialization tables are used in one of the following ways:

- Global variables are initialized at *run time*. Use the --rom_model linker option (see Section 7.10.3.2).
- Global variables are initialized at *load time*. Use the --ram_model linker option (see Section 7.10.3.3).

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 4.1).

For details on linking conventions for EABI with --rom_model and --ram_model, see Section 7.10.4.3 and Section 7.10.4.5, respectively.

For details on linking conventions for COFF with --rom_model and --ram_model, see Section 7.10.3.2 and Section 7.10.3.3, respectively. The following list outlines the linking conventions used for the COFF ABI with --rom_model or --ram_model:

- The symbol _c_int00 is defined as the program entry point; it identifies the beginning of the C/C++ boot routine in boot.c.obj. When you use --rom_model or --ram_model, _c_int00 is automatically referenced, ensuring that boot.c.obj is automatically linked in from the run-time-support library.
- The initialization output section is padded with a termination record so that the loader (load-time initialization) or the boot routine (run-time initialization) knows when to stop reading the initialization tables.
- When initializing at load time (the --ram_model option), the following occur:
  - The linker sets the initialization table symbol to -1. This indicates that the initialization tables are not in memory, so no initialization is performed at run time.
  - The STYP_COPY flag is set in the initialization table section header. STYP_COPY is the special attribute that tells the loader to perform autoinitialization directly and not to load the initialization table into memory. The linker does not allocate space in memory for the initialization table.
- When autoinitializing at run time (--rom_model option), the linker defines the initialization table symbol as the starting address of the initialization table. The boot routine uses this symbol as the starting point for autoinitialization.

4.3.6 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 7.1.1 for a complete description of how the compiler uses these sections.

The compiler creates two basic kinds of sections: initialized and uninitialized. Table 4-1 summarizes the initialized sections. Table 4-2 summarizes the uninitialized sections.

Table 4-1. Initialized Sections

Name	Contents	Restrictions
.args	Reserved space for copying command line arguments before the main() function is called by the boot routine. See Section 2.6.	
.binit	Boot time copy tables (See the *Assembly Language Tools User's Guide* for information on BINIT in linker command files.)	
.c28xabi.exidx	Index table for exception handling; read-only. (EABI only)	Program
.c28xabi.extab	Unwinding instructions for exception handling; read-only. (EABI only)	Program
.cinit	Tables for explicitly initialized global and static variables. (1)	Program
.const	Global and static const variables that are explicitly initialized and contain string literals. String literals are placed in the .const:.string subsection to enable greater link-time placement control. (EABI only)	Data
.data	Global and static non-const variables that are explicitly initialized. (1)	Data
.econst	Global and static const variables that are explicitly initialized and contain string literals. String literals are placed in the .econst:.string subsection to enable greater link-time placement control. (COFF only)	Data
.init_array	Table of constructors to be called at startup. (EABI only)	Program
.ovly	Copy tables other than boot time (.binit) copy tables.	
.pinit	Table of constructors to be called at startup. (COFF only)	Program
.ppdata	Data tables for compiler-based profiling (see the --gen_profile_info option).	Data
.ppinfo	Correlation tables for compiler-based profiling (see the --gen_profile_info option).	Data
.switch	Jump tables for large switch statements.	Program with -mt option Data without -mt option
.text	Executable code and constants.	Program

(1) For EABI, the compiler creates initialized .data sections. The linker creates the .cinit section.

Table 4-2. Uninitialized Sections

Name	Contents	Restrictions
.bss	Global and static variables (EABI only)	Anywhere in data
.ebss	Global and static variables (COFF only)	Anywhere in data
.stack	Stack	Low 64K
.sysmem	Memory for malloc functions (heap) (EABI only)	Anywhere in data
.esysmem	Memory for malloc functions (heap) (COFF only)	Anywhere in data

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.

The linker provides MEMORY and SECTIONS directives for allocating sections. For more information about allocating sections into memory, see the *TMS320C28x Assembly Language Tools User's Guide*.

4.3.7 A Sample Linker Command File

Linker Command File (COFF Example) shows a typical linker command file that links a C program. The command file in this example is named lnk.cmd. It links three object files (x.c.obj, y.c.obj, and z.c.obj) and creates a program (prog.out) and a map file (prog.map).

To link the program, enter the following:

```
cl2000 --run_linker lnk.cmd
```

The MEMORY and possibly the SECTIONS directives, might require modification to work with your system. See the *TMS320C28x Assembly Language Tools User's Guide* for more information on these directives.

Linker Command File (COFF Example)

```
x.c.obj y.c.obj z.c.obj          /* Input filenames */
--output_file=prog.out    /* Options */
--map_file=prog.map
--library=rts2800_ml.lib     /* Get run-time support */
MEMORY                    /* MEMORY directive */
{
   RAM:  origin = 100h     length = 0100h
   ROM:  origin = 01000h   length = 0100h
}
SECTIONS                  /* SECTIONS directive */
{
   .text:    > ROM
   .data:    > ROM
   .ebss:    > RAM
   .pinit:   > ROM
   .cinit:   > ROM
   .switch:  > ROM
   .econst:  > RAM
   .stack:   > RAM
   .esysmem: > RAM
```

If you are using EABI instead of COFF, change the following sections as needed:

- .ebss to .bss
- .esysmem to .sysmem
- .econst to .const
- .pinit to .init_array

4.4 Linking C28x and C2XLP Code

The error in the C28x linker to prevent linking code with a 64-word page size (C28x) and a 128-word page size (C2XLP) has been changed to a warning. It is possible to call a C2XLP assembly function from C28x C/C++ code. One possible way is to replace the call to the C2XLP function with a veneer function that correctly sets up the arguments and call stack for the C2XLP code. For example, to make a call to a C2XLP function expecting five integer arguments, change the C28x code to:

```
extern void foo_veneer(int, int, int, int, int);
void bar()
{
   /* replace the C2XLP call with a veneer call */
   /* foo(1, 2, 3, 4, 5); */
   foo_veneer(1, 2, 3, 4, 5);
}
```

Veneer Function for Linking C2xx and C2XLP Code illustrates how the veneer function might look:

Veneer Function for Linking C2xx and C2XLP Code

```
        .sect ".text"
        .global _foo_veneer
        .global _foo
_foo_veneer:
        ;save registers
        PUSH AR1:AR0
        PUSH AR3:AR2
        PUSH AR5:AR4
        ;set the size of the C2XLP frame (including args size)
        ADDB SP,#10
        ;push args onto the C2XLP frame
        MOV *-SP[10],AL ;copy arg 1
        MOV *-SP[9],AH  ;copy arg 2
        MOV *-SP[8],AR4 ;copy arg 3
        MOV *-SP[7],AR5 ;copy arg 4
        MOV AL,*-SP[19]
        MOV *-SP[6],AL  ;copy arg 5
        ;save the return address
        MOV *-SP[5],#_label
        ;set AR1,ARP
        MOV AL,SP
        SUBB AL,#3
        MOV AR1,AL
        NOP *ARP1
        ;jump to C2XLP function
        LB _foo
_label:
        ;restore register
        POP AR5:AR4
        POP AR3:AR2
        POP AR1:AR0LRETR
```

Since the veneer function frame will act as the frame for all C2XLP calls, it is necessary to add sufficient size to the frame for any subsequent calls made by the first C2XLP function.

Global variables will be placed in the .ebss sections for C28x C/C++ code compiled for the COFF ABI. A C2XLP .ebss section is not guaranteed to begin on a 128-word boundary when linked with C28x code. To avoid this problem, define a new section, change the C2XLP globals to the new section, and update the linker command file to ensure this new section begins at a 128-word boundary.

Chapter 5

Post-Link Optimizer

The TMS320C28x post-link optimizer removes or modifies assembly language instructions to generate better code. The post-link optimizer examines the final addresses of symbols determined by linking and uses this information to make code changes.

Post-link optimization requires the -plink compiler option. The -plink compiler option invokes added passes of the tools that include running the absolute lister and rerunning the assembler and linker. You must use the -plink option following the --run_linker option.

5.1 The Post-Link Optimizer's Role in the Software Development Flow

The post-link optimizer is not part of the normal development flow. Figure 5-1 shows the flow including the post-link optimizer; this flow occurs only when you use the compiler and the -plink option.

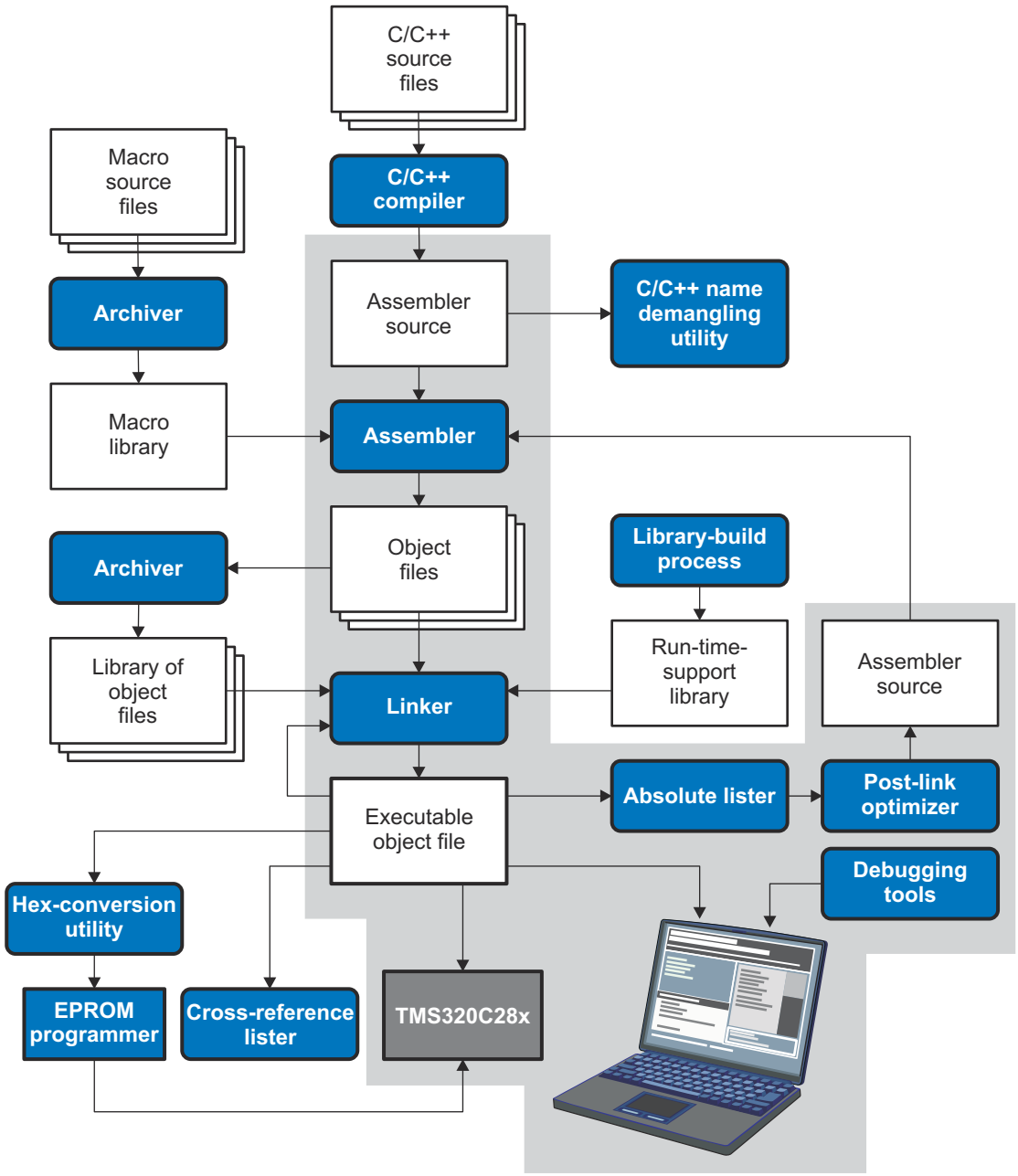


Figure 5-1. The Post-Link Optimizer in the TMS320C28x Software Development Flow

As the flow shows, the absolute lister (abs2000) is also part of the post-link optimizing process. The absolute lister outputs the absolute addresses of all globally defined symbols and COFF sections. The post-link optimizer takes .abs files as input and uses these addresses to perform optimizations. The output is a .pl file, which is an optimized version of the original .asm file. The flow then reruns the assembler and linker to produce a final output file.

The described flow is supported only when you use the compiler (cl2000) and the -plink option. If you use a batch file to invoke each tool individually, you must adapt the flow to use the compiler instead. In addition, you

must use the --output_file option to specify an output file name when using the -plink option. See Section 5.8 for more details.

For example, replace these lines:

```
cl2000 file1.asm file1.c.obj
cl2000 file2.asm file2.c.obj
cl2000 --run_linker file1.c.obj file2.c.obj lnk.cmd --output_file=prog.out
```

with this line:

```
cl2000 file1.asm file2.asm --run_linker lnk.cmd --output_file=prog.out -plink
```

The advice_only mode is the only post-link optimization support provided for FPU and VCU targets.

5.2 Removing Redundant DP Loads

Post-link optimization reduces the difficulty of managing the Data Page (DP) register by removing redundant DP loads. It does this by tracking the current value of the DP and determining whether the address in a MOV DP,#address instruction is located on the same 64-word page to which the DP is currently pointing. If the address can be accessed using the current DP value, the instruction is redundant and can be removed. For example, consider the following code segment:

```
    MOVZ      DP,#name1
    ADD       @name1,#10
    MOVZ      DP,#name2
    ADD       @name2,#10
```

If name1 and name2 are linked to the same page, the post-link optimizer determines that loading DP with the address of name2 is not necessary, and it comments out the redundant load.

```
    MOVZ      DP,#name1
    ADD       @name1,#10
     ; <<REDUNDANT>>          MOVZ      DP,#name2
    ADD       @name2,#10
```

This optimization can be used on C files as well. Even though the compiler manages the DP for all global variable references that are defined within a module, it conservatively emits DP loads for any references to global variables that are externally defined. Using the post-link optimizer can help reduce the number of DP loads in these instances.

Additionally, the --map_file linker option can be used to generate a listing of symbols sorted by data page to assist with data layout and DP load reduction. For more information, refer to the Linker Description chapter in the *TMS320C28x Assembly Language Tools User's Guide*.

5.3 Tracking DP Values Across Branches

In order to track DP values across branches, the post-link optimizer requires that there are no indirect calls or branches, and all possible branch destinations have labels. If an indirect branch or call is encountered, the post-link optimizer will only track the DP value within a basic block. Branch destinations without labels may cause incorrect output from the post-link optimizer.

If the post-link optimizer encounters indirect calls or branches, it issues the following warning:

```
NO POST LINK OPTIMIZATION DONE ACROSS BRANCHES
Branch/Call must have labeled destination
```

This warning is issued so that if the file is a hand written assembly file, you can try to change the indirect call/branch to a direct one to obtain the best optimization from the post linker.

5.4 Tracking DP Values Across Function Calls

The post-link optimizer optimizes DP loads after a call to a function if the function is defined in the same file scope. For example, consider the following post-link optimized code:

```
_main:
    LCR    #_foo
    MOVB    AL, #0
;<<REDUNDANT>>    MOVZ    DP,#_g2
    MOV    @_g2, #20
    LRETR
    .global    _foo
_foo:
    MOVZ    DP, #g1
    MOV      @_g1, #10
    LRETR
```

The MOVZ DP after the function call to _foo is removed by the post-link optimizer as the variables _g1 and _g2 are in the same page and the function _foo already set the DP.

In order for the post-link optimizer to optimize across the function calls, the functions should have only one return statement. If you are running the post-link optimizer on hand written assembly that has more than one return statement per function, the post-link optimization output can be incorrect. You can turn off the optimization across function calls by specifying the -nf option after the -plink option.

5.5 Other Post-Link Optimizations

An externally defined symbol used as a constant operand forces the assembler to choose a 16-bit encoding to hold the immediate value. Since the post-link optimizer has access to the externally defined symbol value, it replaces a 16-bit encoding with an 8-bit encoding when possible. For example:

```
    .ref ext_sym ; externally defined to be 4
    :
    :
    ADD AL, #ext_sym ; assembly will encode ext_sym with 16 bits
```

Since ext_sym is externally defined, the assembler chooses a 16-bit encoding for ext_sym. The post-link optimizer changes the encoding of ext_sym to an 8-bit encoding:

```
    .ref ext_sym
    :
    :
    ; << ADD=>ADDB>> ADD AL,#ext_sym
    ADDB AL, #ext_sym
```

Similarly the post-link optimizer attempts to reduce the following 2-word instructions to 1-word instructions:

2-Word Instructions	1-Word Instructions
ADD ACC, *#imm*	ADDB ACC, *#imm*
ADD AL, *#imm*	ADDB AL, *#imm*
AND AL, *#imm*	ANDB AL, *#imm*
CMP AL, *#imm*	CMPB AL, *#imm*
MOVL XARn, *#imm*	MOVB XARn, *#imm*
OR AL, *#imm*	ORB AL, *#imm*
SUB ACC, *#imm*	SUBB ACC, *#imm*
SUB AL, *#imm*	SUBB AL, *#imm*
XOR AL, *#imm*	XORB AL, *#imm*

5.6 Controlling Post-Link Optimizations

There are three ways to control post-link optimizations: by excluding files, by inserting specific comments within an assembly file, and by manually editing the post-link optimization file.

5.6.1 Excluding Files (-ex Option)

Specific files can be excluded from the post-link optimization process by using the −ex option. The files to be excluded must follow the −ex option and include file extensions. The −ex option must be used after the −plink option and no other option may follow. For example:

```
cl2000 file1.asm file2.asm file3.asm --keep_asm --run_linker lnk.cmd -plink -o=prog.out -
ex=file3.asm
```

The file3.asm will be excluded from the post-link optimization process.

5.6.2 Controlling Post-Link Optimization Within an Assembly File

Within an assembly file, post-link optimizations can be disabled or enabled by using two specially formatted comment statements:

```
;//NOPLINK//
;//PLINK//
```

Assembly statements following the NOPLINK comment are not optimized. Post-link optimization can be reenabled using the //PLINK// comment.

The PLINK and NOPLINK comment format is not case sensitive. There can be white space between the semicolon and PLINK delimiter. The PLINK and NOPLINK comments must appear on a separate line, alone, and must begin in column 1. For example:

```
;      //PLINK//
```

5.6.3 Retaining Post-Link Optimizer Output (--keep_asm Option)

The --keep_asm option allows you to retain any post-link files (.pl) and .absolute listing files (.abs) generated by the −plink option. Using the --keep_asm option lets you view any changes the post-link optimizer makes.

The .pl files contain the commented out statement shown with <<REDUNDANT>> or any improvements to instructions, such as <<ADD=>ADDB>>. The .pl files are assembled and linked again to exclude the commented out lines.

5.6.4 Disable Optimization Across Function Calls (-nf Option)

The −nf option disables the post-link optimization across function calls. The post-link optimizer recognizes the function end by the return statement and assumes there is only one return statement per function. In some hand written assembly code, it is possible to have more than one return statement per function. In such cases, the output of the post-link optimization can be incorrect. You can turn off the optimization across function calls by using the− nf option. This option affects all the files.

5.6.5 Annotating Assembly with Advice (--plink_advice_only option)

The --plink_advice_only option annotates assembly code with comments if changes cannot be made safely due to pipeline considerations, such as when float support or VCU support is enabled.

If you use this option, note that the post-link files (.pl), which contain the generated advice, are retained only if you also use the --keep_asm option.

5.7 Restrictions on Using the Post-Link Optimizer

The following restrictions affect post-link optimization:

- The advice_only mode is the only post-link optimization support provided for FPU and VCU targets.
- Branches or calls to unlabeled destinations invalidate DP load optimizations. All branch destinations must have labels.
- If the position of the data sections depends on the size of the code sections, the data page layout information used to decide which DP load instructions to remove may no longer be valid.

  For example, consider the following link command file:

```
SECTIONS
{
    .text    > MEM,
    .mydata  > MEM,}
```

  A change in the size of the .text section after optimizing causes the .mydata section to shift. Ensuring that all output data sections are aligned on a 64-word boundary removes this shifting issue. For example, consider the following link command file:

```
SECTIONS
{
    .text > MEM,
    .mydata align = 64 > MEM,}
```

5.8 Naming the Outfile (--output_file Option)

When using the -plink option, you must include the --output_file option. If the output filename is specified in a linker command file, the compiler does not have access to the filename to pass it along to other phases of post-link optimization, and the process will fail. For example:

```
cl2000 file1.c file2.asm --run_linker --output_file=prog.out lnk.cmd -plink
```

Because the post-link optimization flow uses the absolute lister, abs2000, it must be included in your path.

Chapter 6

C/C++ Language Implementation

The C language supported by the C28x 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 C28x is defined by the ANSI/ISO/IEC 14882:2003 standard with certain exceptions.

6.1 Characteristics of TMS320C28x 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 6.15).

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 6.14.

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 int (16 bits) for COFF and long (32 bits) for EABI. 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 6.4 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 6.14.

6.1.1 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 6.4 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 6.4 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 16. See Section 6.3 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 6.4 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

- No extended integer types are provided. (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)
- No extended integer types are provided, so there is no change to the integer ranks. (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 an integer is converted to a floating-point number is IEEE-754 "round to nearest". (6.3.1.4)
- 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. Some pointers are not the same size as any integral type, but the conversion proceeds as if such a type did exist, with the rules implied by normal integer conversion.
- 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 6.3. (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 2.11.2. (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. (6.7.2.1)
- Non-bit-field members of structures are aligned as specified in Section 7.1.7. (6.7.2.1)
- The integer type underlying each enumerated type is described in Section 6.3.1. (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 2.5.2. (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 2.5.2. (6.10.2).
- he compiler uses the file search path to search for an included " " delimited header file. See Section 2.5.2. (6.10.2). (6.10.2).
- There is no arbitrary nesting limit for #include processing. (6.10.2).
- See Section 6.9 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 6.14.1. (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 6.3. (5.2.4.2, 7.18.2, 7.18.3)
- The number, order, and encoding of bytes in any object are described in . (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 6.3. (6.5.3.4)

6.2 Characteristics of TMS320C28x C++

The C28x compiler supports C++ as defined in the ANSI/ISO/IEC 14882:2003 standard (C++03), including these features:

- Complete C++ standard library support, with exceptions noted below.
- Templates
- Exceptions, which are enabled with the --exceptions option; see Section 6.6.
- Run-time type information (RTTI), which can be enabled with the --rtti compiler option.

The compiler supports the 2003 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.
- If the definition of an inline function contains a static variable, and it appears in multiple compilation units (usually because it's a member function of a class defined in a header file), the compiler generates multiple copies of the static variable rather than resolving them to a single definition. The compiler emits a warning (#1369) in such cases.
- The export keyword is not implemented.

6.3 Data Types

Table 6-1 lists the size, representation, and range of each scalar data type for the C28x compiler. Many of the range values are available as standard macros in the header file limits.h.

Types with a size of 16 bits are aligned on 16-bit boundaries. Types with a size of 32 bits or larger are aligned on 32-bit (2 word) boundaries. For details on EABI field alignment, refer to the *C28x Embedded Application Binary Interface (EABI) Reference Guide* (SPRAC71). The storage and alignment of data types is further described in Section 7.1.7.

Table 6-1. TMS320C28x C/C++ COFF and EABI Data Types

			Range	
Type	**Size**	**Representation**	**Minimum**	**Maximum**
char, signed char	16 bits	ASCII	-32 768	32 767
unsigned char	16 bits	ASCII	0	65 535
_Bool	16 bits	Binary	0 (false)	1 (true)
short	16 bits	Binary	-32 768	32 767
unsigned short	16 bits	Binary	0	65 535
int, signed int	16 bits	Binary	-32 768	32 767
unsigned int	16 bits	Binary	0	65 535
long, signed long	32 bits	Binary	-2 147 483 648	2 147 483 647
unsigned long	32 bits	Binary	0	4 294 967 295
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 (1)	varies	Binary	varies	varies
float (2)	32 bits	IEEE 32-bit	1.19 209 290e-38(3)	3.40 282 35e+38
double (COFF)	32 bits	IEEE 32-bit	1.19 209 290e-38(3)	3.40 282 35e+38 (COFF)
double (EABI)	64 bits	IEEE 64-bit	2.22 507 385e-308(3)	1.79 769 313e+308
long double	64 bits	IEEE 64-bit	2.22 507 385e-308(3)	1.79 769 313e+308
pointers (4)	32 bits	Binary	0	0xFFFFFFFF

(1) For details about the size of an enum type, see Section 6.3.1.
(2) It is recommended that 32-bit floating point values for COFF be declared as `float` , not as `double` .
(3) Figures are minimum precision.
(4) Even though pointers are 32-bits, the compiler assumes that the addresses of global variables and functions are within 22-bits.

The wchar_t type is implemented as int (16 bits) for COFF and long (32 bits) for EABI.

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

Note

TMS320C28x Byte is 16 Bits

By ANSI/ISO C definition, the sizeof operator yields the number of bytes required to store an object. ANSI/ISO further stipulates that when sizeof is applied to char, the result is 1. Since the TMS320C28x char is 16 bits (to make it separately addressable), a byte is also 16 bits. This yields results you may not expect; for example, size of (int) = = 1 (not 2). TMS320C28x bytes and words are equivalent (16 bits). To access data in increments of 8 bits, use the __byte() and __mov_byte() intrinsics described in Section 7.6.

6.3.1 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 the following list that is big enough and of the correct sign to represent all of the values of the enumeration constants. The "char" and "short" types are not used because they are the same size as the "int" type.

- unsigned int
- signed int
- unsigned long
- signed long
- unsigned long long
- signed long long

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

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

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

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

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

```
enum sl { a, b, c, d = -1, e = UINT_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 don't 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.

6.3.2 Support for 64-Bit Integers

The TMS320C28x compiler supports the long long and unsigned long long data types. The range values are available as standard macros in the header file limits.h.

The long long data types are stored in register pairs. In memory they are stored as 64-bit objects at double-word (32-bit) aligned addresses.

A long long integer constant can have an ll or LL suffix. Without the suffix the value of the constant determines the type of the constant.

The formatting rules for long long in C I/O require ll in the format string. For example:

```
printf("%lld", 0x0011223344556677);
printf("%llx", 0x0011223344556677);
```

The run-time-support library functions, llabs(), strtoll() and strtoull(), are added.

6.3.3 C28x double and long double Floating-Point Types

When compiling C/C++ code for the TMS320C28x, the `long double` floating point type uses the IEEE 64-bit double precision format.

The `double` type differs depending on whether you are compiling for COFF or EABI. For COFF, it uses the IEEE 32-bit single precision type. For EABI it uses IEEE 64-bit double precision.

C28x floating point types are:

Type	Format
float	IEEE 32-bit single precision
double (COFF)	IEEE 32-bit single precision
double (EABI)	IEEE 64-bit double precision
long double	IEEE 64-bit double precision

It is recommended that 32-bit floating point values be declared as `float` , not as `double` .

When you initialize a long double to a constant, you must use the l or L suffix. The constant is treated as a double type without the suffix and the run-time support double-to-long conversion routine is called for the initialization. This could result in the loss of precision. For example:

```
long double a = 12.34L; /* correctly initializes to double precision */
long double b = 56.78;  /* converts single precision value to double precision */
```

The formatting rules for long doubles in C I/O require a capital 'L' in the format string. For example:

```
printf("%Lg", 1.23L);
printf("%Le", 3.45L);
```

For EABI mode, see the *C28x Embedded Application Binary Interface Application Report* (SPRAC71) for information on calling conventions for 64-bit types.

For COFF, the calling conventions for the long double type are as follows:

All long double arguments are passed by reference. A long double return value is returned by reference. The first two long double arguments will pass their addresses in XAR4 and XAR5. All other long double arguments will have their addresses passed on the stack.

If a function returns a long double, the function making that call will place the return address in XAR6. For example:

```
long double foo(long double a, long double b, long double c)
{
   long double d = a + b + c;
   return d;
}
long double a = 1.2L;
long double b = 2.2L;
long double c = 3.2L;
long double d;
void bar()
{
   d = foo(a, b, c);
}
```

In function bar(), at the call to foo(), the register values are:

Register	Equals
XAR4	The address of a
XAR5	The address of b
*−SP[2]	The address of c
XAR6	The address of d

The run-time-support library includes the necessary long double arithmetic operations and conversion functions.

6.4 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.

6.5 Keywords

The C28x 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,__cregister,and __asm. The compiler supports the restrict keyword for FPU targets; for other targets restrict is ignored. 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

6.5.1 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 const are placed in the .econst or .const section. The linker allocates the .econst or .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 (function scope).
- 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:

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

6.5.2 The __cregister Keyword

The compiler extends the C/C++ language by adding the cregister keyword to allow high level language access to control registers. This keyword is available in normal mode, but not in strict ANSI/ISO mode (using the --strict_ansi compiler option). The alternate keyword, __cregister, provides the same functionality but is available in either strict ANSI/ISO mode or normal mode.

When you use the cregister keyword on an object, the compiler compares the name of the object to a list of standard control registers for the C28x (see Table 6-2). If the name matches, the compiler generates the code to reference the control register. If the name does not match, the compiler issues an error.

Table 6-2. Valid Control Registers

Register	Description
IER	Interrupt enable register
IFR	Interrupt flag register

The cregister keyword can be used only in file scope. The cregister keyword is not allowed on any declaration within the boundaries of a function. It can only be used on objects of type integer or pointer. The cregister keyword is not allowed on objects of any floating-point type or on any structure or union objects.

The cregister keyword does not imply that the object is volatile. If the control register being referenced is volatile (that is, can be modified by some external control), then the object must be declared with the volatile keyword also.

To use the control registers in Table 6-2, you must declare each register as follows. The c28x.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, though in a limited manner. IFR is read only and can be set only by using the | (OR) operation with an immediate. IFR can be cleared only by using the & (AND) operation with an immediate. For example:

```
IFR |= 0x4;
IFR &= 0x0800
```

The IER register also can be used in an assignment other than OR and AND. Since the C28x architecture has limited instructions to manipulate these registers, the compiler terminates with the following message if illegal use of these registers is found:

```
>>> Illegal use of control register
```

See Define and Use Control Registers for an example that declares and uses control registers.

Define and Use Control Registers

```
extern cregister volatile unsigned int IFR;
extern cregister volatile unsigned int IER;
extern int x;
main()
{
   IER = x;
   IER |= 0x100;
   printf("IER = %x\n", IER);
   IFR &= 0x100;
   IFR |= 0x100;
```

6.5.3 The __interrupt Keyword

The compiler extends the C/C++ language by adding the __interrupt keyword, which specifies that a function is treated as an interrupt function. 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 6.9.15 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.

6.5.4 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 6.14.

The following example uses the restrict keyword 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;
  }
}
```

At this time the restrict keyword is useful only for FPU targets. For non-FPU targets restrict is ignored.

6.5.5 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 --unified_memory option can be used if your memory map is configured as a single unified space; this option allows the compiler to generate more efficient instructions for most memcpy calls and structure assignments. Even under unified memory, memory for some peripherals and some RAM associated with those peripherals is allocated only in data memory. If –unified_memory is enabled, you can prevent program memory address access to specific symbols by declaring those symbols as volatile.

6.6 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 4.3.1.1. 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 and execution time, even if no exceptions are thrown. This is true particularly for the COFF ABI.

See Section 8.1 for details on the run-time libraries.

6.7 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 7.2.

6.8 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 6.15.2.

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 *TMS320C28x 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 7.5.5.

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.

Note

Use Single asm Statement for the RPT Instruction

When adding a C28x RPT instruction, do not use a separate asm statement for RPT and the repeated instruction. The compiler could insert debug directives between asm directives and the assembler does not allow any directives between the RPT and the repeated instruction. For example, to insert a RPT MAC instruction, use the following:

```
asm("\tRPT #10\n\t||MAC P, *XAR4++, *XAR7++");
```

6.9 Pragma Directives

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

- CALLS (See Section 6.9.1)
- CLINK (See Section 6.9.2)
- CODE_ALIGN (See Section 6.9.3)
- CODE_SECTION (See Section 6.9.4)
- DATA_ALIGN (See Section 6.9.5)
- DATA_SECTION (See Section 6.9.6)
- diag_suppress, diag_remark, diag_warning, diag_error, diag_default, diag_push, diag_pop (See Section 6.9.7)
- FAST_FUNC_CALL (See Section 6.9.8)
- FORCEINLINE (See Section 6.9.9)
- FORCEINLINE_RECURSIVE (See Section 6.9.10)
- FUNC_ALWAYS_INLINE (See Section 6.9.11)
- FUNC_CANNOT_INLINE (See Section 6.9.12)
- FUNC_EXT_CALLED (See Section 6.9.13)
- FUNCTION_OPTIONS (See Section 6.9.14)
- INTERRUPT (See Section 6.9.15)
- LOCATION (See Section 6.9.16)
- MUST_ITERATE (See Section 6.9.17)
- NOINIT (See Section 6.9.18)
- NOINLINE (See Section 6.9.19)
- NO_HOOKS (See Section 6.9.20)
- once (See Section 6.9.21)
- PERSISTENT (See Section 6.9.18)
- RETAIN (See Section 6.9.22)
- SET_CODE_SECTION (See Section 6.9.23)
- SET_DATA_SECTION (See Section 6.9.23)
- UNROLL (See Section 6.9.24)
- WEAK (See Section 6.9.25)

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 the pragma specification must occur before any declaration, definition, or reference to the func or symbol argument. If you do not follow these rules, the compiler issues a warning 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.

6.9.1 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 *TMS320C28x 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 6.15.2), which has the same effect as the CALLS pragma, has the following syntax:

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

6.9.2 The CLINK Pragma

Note

The CLINK pragma is used in COFF mode. It has no effect when used in EABI mode because conditional linking is enabled by default.

The CLINK pragma can be applied to a code or data symbol. It causes a .clink directive to be generated into the section that contains the definition of the symbol. The .clink directive tells the linker that a section is eligible for removal during conditional linking. Thus, if the section is not referenced by any other section in the application being compiled and linked, it will not be included in the resulting output file.

The syntax of the pragma in C is:

#pragma CLINK (*symbol* **)**

The syntax of the pragma in C++ is:

#pragma CLINK

The RETAIN pragma has the opposite effect of the CLINK pragma. See Section 6.9.22 for more details.

6.9.3 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 6.15.2.

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* **)**

6.9.4 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 6.15.2.

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

```
char bufferA[80];
char bufferB[80];
#pragma CODE_SECTION(funcA, "codeA")
char funcA(int i);
char funcB(int i);
void main()
{
   char c;
   c = funcA(1);
   c = funcB(2);
}
char funcA (int i)
{
   return bufferA[i];
}
char funcB (int j)
{
   return bufferB[j];
}
```

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

```
        .sect   ".text"
        .global _main;
*****************************************************************
;* FNAME: _main                          FR SIZE:   2           *
;*                                                               *
;* FUNCTION ENVIRONMENT                                          *
;*                                                               *
;* FUNCTION PROPERTIES                                           *
;*                           0 Parameter,  1 Auto,  0 SOE        *
;*****************************************************************
:_main:
        ADDB      SP,#2
        MOVB      AL,#1                 ; |12|
        LCR       #_funcA               ; |12|
               ; call occurs [#_funcA] ; |12|
        MOV       *-SP[1],AL            ; |12|
        MOVB      AL,#1                 ; |13|
        LCR       #_funcB               ; |13|
                ; call occurs [#_funcB] ; |13|
        MOV       *-SP[1],AL            ; |13|
        SUBB      SP,#2
        LRETR
        ; return occurs
        .sect   "codeA"
        .global _funcA
;*****************************************************************
;* FNAME: _funcA                         FR SIZE:   1           *
;*                                                               *
;* FUNCTION ENVIRONMENT                                          *
;*                                                               *
;* FUNCTION PROPERTIES                                           *
;*                           0 Parameter,  1 Auto,  0 SOE        *
;*****************************************************************
_funcA:
        ADDB      SP,#1
        MOV       *-SP[1],AL            ; |17|
        MOVZ      AR6,*-SP[1]           ; |18|
        ADD       AR6,#_bufferA         ; |18|
        SUBB      SP,#1                 ; |18|
        MOV       AL,*+XAR6[0]          ; |18|
        LRETR
        ;return occurs
         .sect   ".text"
         .global _funcB;
*****************************************************************
;* FNAME: _funcB                         FR SIZE:   1           *
;*                                                               *
;* FUNCTION ENVIRONMENT                                          *
;*                                                               *
;* FUNCTION PROPERTIES                                           *
;*                           0 Parameter,  1 Auto,  0 SOE        *
;*****************************************************************
_funcB:
        ADDB      SP,#1
        MOV       *-SP[1],AL            ; |22|
        MOVZ      AR6,*-SP[1]           ; |23|
        ADD       AR6,#_bufferB         ; |23|
        SUBB      SP,#1                 ; |23|
        MOV       AL,*+XAR6[0]          ; |23|
        LRETR
        ;return occurs
```

6.9.5 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 6.15.4.

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* **)**

6.9.6 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 .ebss or .bss section.

Using the DATA_SECTION pragma has the same effect as using the GCC-style `section` variable attribute. See Section 6.15.4.

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 6-1 through Example 6-3 demonstrate the use of the DATA_SECTION pragma.

Example 6-1. Using the DATA_SECTION Pragma C Source File

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

Example 6-2. Using the DATA_SECTION Pragma C++ Source File

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

Example 6-3. Using the DATA_SECTION Pragma Assembly Source File

```
            .global _bufferA
            .ebss    _bufferA,512,4
            .global _bufferB
_bufferB:   .usect  "my_sect",512,4
```

6.9.7 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.

6.9.8 The FAST_FUNC_CALL Pragma

The FAST_FUNC_CALL pragma, when applied to a function, generates a TMS320C28x FFC instruction to call the function instead of the CALL instruction. Refer to the *TMS320C28x DSP CPU and Instruction Set User's Guide* for more details on the FFC instruction.

The syntax of the pragma in C is:

#pragma FAST_FUNC_CALL (*func* **)**

The syntax of the pragma in C++ is:

#pragma FAST_FUNC_CALL (*func* **)**

The FAST_FUNC_CALL pragma should be applied only to a call to an assembly function that returns with the LB *XAR7 instruction. See Section 7.5.1 for information on combining C/C++ and assembly code.

Since this pragma should be applied only to assembly functions, if the compiler finds a definition for *func* in the file scope, it issues a warning and ignores the pragma.

The following example demonstrates the use of the FAST_FUNC_CALL pragma.

Using the FAST_FUNC_CALL Pragma Assembly Function

```
_add_long:
        ADD ACC, *-SP[2]
        LB *XAR7
```

Using the FAST_FUNC_CALL Pragma C Source File

```
#pragma FAST_FUNC_CALL (add_long)
long add_long(long, long);
void foo()
{
   long x, y;
   x = 0xffff;
   y = 0xff;
   y = add_long(x, y);
}
```

Generated Assembly File

```
;****************************************************************
;* FNAME: _foo                          FR SIZE:   6            *
;*                                                              *
;* FUNCTION ENVIRONMENT                                         *
;*                                                              *
;* FUNCTION PROPERTIES                                          *
;*                            2 Parameter,  4 Auto,  0 SOE      *
;****************************************************************
_foo:
        ADDB      SP,#6
        MOVB      ACC,#255
        MOVL      XAR6,#65535           ; |8|
        MOVL      *-SP[6],ACC
        MOVL      *-SP[2],ACC           ; |10|
        MOVL      *-SP[4],XAR6          ; |8|
        MOVL      ACC,*-SP[4]           ; |10|
        FFC       XAR7,#_add_long       ; |10|
        ; call occurs [#_add_long]      ; |10|
        MOVL      *-SP[6],ACC           ; |10|
        SUBB      SP,#6
        LRETR
        ; return occurs
```

6.9.9 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 2.11.

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.

6.9.10 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 2.11.

6.9.11 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 2.11 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 6.15.2.

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.

6.9.12 The FUNC_CANNOT_INLINE Pragma

The FUNC_CANNOT_INLINE pragma instructs the compiler that the named function cannot be expanded inline. Any function named with this pragma overrides any inlining you designate in any other way, such as using the inline keyword. Automatic inlining is also overridden with this pragma; see Section 2.11.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that cannot be inlined. In C++, the pragma applies to the next function declared.

The FUNC_CANNOT_INLINE pragma has the same effect as using the GCC-style `noinline` function attribute. See Section 6.15.2.

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

6.9.13 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 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 3.4.2.

6.9.14 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.

6.9.15 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 **)**

The GCC interrupt attribute syntax, which has the same effects as the INTERRUPT pragma, is as follows. Note that the interrupt attribute can precede either the function's definition or its declaration.

__attribute__((interrupt)) void *func* **(** void **)**

On the FPU, there are two kinds of interrupts - High Priority Interrupt (HPI) and Low Priority Interrupt (LPI). High priority interrupts use a fast context save and cannot be nested. Low priority interrupts behave like normal C28x interrupts and can be nested.

The kind of interrupt can be specified by way of the interrupt pragma using an optional second argument. The C syntax of the pragma is:

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

The syntax of the pragma in C++ is:

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

The syntax of the GCC interrupt attribute, which has the same effects as the INTERRUPT pragma, is:

__attribute__((interrupt("HPI"|"LPI"))) void *func* **(** void **)**
{ ... }

On FPU, if no interrupt priority is specified LPI is assumed. Interrupts specified with the interrupt keyword also default to LPI.

CLA interrupts and CLA2 background tasks can be created using the interrupt attribute or the INTERRUPT pragma. For example the following could both be used with a CLA interrupt:

```
__attribute__((interrupt))
void interrupt_name(void) {...}
#pragma INTERRUPT(interrupt _name);
void interrupt _name(void) {...}
```

The following examples both create a CLA2 background task:

```
__attribute__((interrupt("BACKGROUND")))
 void task_name(void) {...}
#pragma INTERRUPT(task_name, "BACKGROUND");
void task_name(void) {...}
```

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.

6.9.16 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 6.15.2.

Note

This pragma is supported only when used with EABI. It is not supported with the COFF ABI.

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 6.15.4) 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 6.9.18.

6.9.17 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, 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, can appear between the MUST_ITERATE pragma and the loop.

6.9.17.1 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];
    }
    ...
}
```

6.9.17.2 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 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 loop:

```
#pragma MUST_ITERATE(8, 8)
for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

6.9.18 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.

Note

These pragmas are supported only when used with EABI. They are not supported with the COFF ABI.

Variables can be declared as noinit or persistent using either pragmas or variable attributes. See Section 6.15.4 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;
```

6.9.19 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 2.11.

6.9.20 The NO_HOOKS Pragma

The NO_HOOKS pragma prevents entry and exit hook calls from being generated for a function.

The syntax of the pragma in C is:

```
#pragma NO_HOOKS ( func )
```

The syntax of the pragma in C++ is:

```
#pragma NO_HOOKS
```

See Section 2.14 for details on entry and exit hooks.

6.9.21 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.

6.9.22 The RETAIN Pragma

The RETAIN pragma can be applied to a code or data symbol.

In EABI mode, which assumes that all sections are *eligible* for removal via conditional linking, this pragma 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.

In COFF mode, which assumes that all sections are *ineligible* for removal, the RETAIN pragma prevents the compiler from issuing a .clink directive for a symbol.

The RETAIN pragma has the same effect as using the `retain` function or variable attribute. See Section 6.15.2 and Section 6.15.4, respectively.

The syntax of the pragma in C is:

#pragma RETAIN (*symbol* **)**

The syntax of the pragma in C++ is:

#pragma RETAIN

The CLINK pragma has the opposite effect of the RETAIN pragma and is used only in COFF mode. See Section 6.9.2 for more details.

6.9.23 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.

6.9.24 The UNROLL Pragma

The UNROLL pragma specifies to the compiler how many times a loop should be unrolled. 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, 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 6.9.17.1 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.

6.9.25 The WEAK Pragma

The WEAK pragma gives weak binding to a symbol.

Note

This pragma is supported only when used with EABI. It is not supported with the COFF ABI.

The syntax of the pragma in C is:

```
#pragma WEAK ( symbol )
```

The syntax of the pragma in C++ is:

```
#pragma WEAK
```

The WEAK pragma makes *symbol* a weak reference if it is a reference, or a weak definition, if it is a definition. The symbol can be a data or function variable. In effect, unresolved weak *references* do not cause linker errors and do not have any effect at run time. The following apply for weak references:

- Libraries are not searched to resolve weak references. It is not an error for a weak reference to remain unresolved.
- During linking, the value of an undefined weak reference is:
  - Zero if the relocation type is absolute
  - The address of the place if the relocation type is PC-relative
  - The address of the nominal base address if the relocation type is base-relative.

A weak *definition* does not change the rules by which object files are selected from libraries. However, if a link set contains both a weak definition and a non-weak definition, the non-weak definition is always used.

The WEAK pragma has the same effect as using the `weak` function or variable attribute. See Section 6.15.2 and Section 6.15.4, respectively.

6.10 The _Pragma Operator

The C28x 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.

6.11 Application Binary Interface

An Application Binary Interface (ABI) defines how functions that are compiled or assembled separately (possibly by compilers from different vendors) can work together. This involves standardizing the data type representation, register conventions, and function structure and calling conventions. An ABI defines linkname generation from C symbol names. It defines the object file format and the debug format. It defines how the system is initialized. In the case of C++, it defines C++ name mangling and exception handling support.

The TI C28x Code Generation Tools support both the COFF ABI and the EABI ABI. The default is to generate object files using COFF. Selecting the ABI is controlled by the --abi command-line option and is discussed in Section 2.13. The __TI_EABI__ predefined symbol is defined and set to 1 if the code is compiled for EABI.

To generate object files compatible with EABI, you must use the C28x compiler version 18.8.0.STS or greater. The COFF ABI is the only ABI supported by older compilers.

The EABI ABI requires the ELF object file format. This format supports modern language features like early template instantiation and exporting inline functions. For low-level details about the C28x EABI, see the *C28x Embedded Application Binary Interface Application Report* (SPRAC71).

All of the object files linked to create an application must use the same ABI. That is, they must all use the COFF ABI or all use EABI; these ABIs are not compatible with each other.

6.12 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.

For EABI, 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.

For COFF, the compiler places an underscore at the beginning of the linknames of C identifiers, so you can safely use identifiers that do not begin with an underscore in your assembly code.

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.

The COFF mangling algorithm used closely follows that described in The Annotated Reference Manual (ARM).

For example, the general form of a C++ linkname for a function named func is:

_func__F *parmcodes*

Where parmcodes is a sequence of letters that encodes the parameter types of func.

For this simple C++ source file:

```
int foo(int i){ }   //global C++ function
```

This is the resulting assembly code:

```
_foo__Fi
```

The linkname of foo is _foo__Fi, indicating that foo is a function that takes a single argument of type int. To aid inspection and debugging, a name demangling utility is provided that demangles names into those found in the original C++ source. See Chapter 9 for more information.

6.13 Initializing Static and Global Variables in COFF ABI Mode

The ANSI/ISO C standard specifies that global (extern) and static variables without explicit initializations must be initialized to 0 before the program begins running. This task is typically done when the program is loaded. Because the loading process is heavily dependent on the specific environment of the target application system, in COFF ABI mode the compiler itself makes no provision for initializing to 0 otherwise uninitialized static storage class variables at run time. It is up to your application to fulfill this requirement.

Note

Initialize Global Objects

You should explicitly initialize all global objects which you expected the compiler would set to zero by default.

In EABI mode the uninitialized variables are zero initialized automatically.

6.13.1 Initializing Static and Global Variables With the Linker

If your loader does not preinitialize variables, you can use the linker to preinitialize the variables to 0 in the object file. For example, in the linker command file, use a fill value of 0 in the .ebss section:

```
SECTIONS
   {
      ...
      .ebss: {} = 0x00;
      ...
   }
```

Because the linker writes a complete load image of the zeroed .ebss section into the output COFF file, this method can have the unwanted effect of significantly increasing the size of the output file (but not the program).

If you burn your application into ROM, you should explicitly initialize variables that require initialization. The preceding method initializes .ebss to 0 only at load time, not at system reset or power up. To make these variables 0 at run time, explicitly define them in your code.

For more information about linker command files and the SECTIONS directive, see the linker description information in the *TMS320C28x Assembly Language Tools User's Guide.*

6.13.2 Initializing Static and Global Variables With the const Type Qualifier

Static and global variables of type *const* without explicit initializations are similar to other static and global variables because they might not be preinitialized to 0 (for reasons discussed in Section 6.13). For example:

```
const int zero;      /*  might not be initialized to 0  */
```

However, the initialization of const global and static variables is different because these variables are declared and initialized in a section called .econst or .const (depending on the ABI). For example:

```
const int zero = 0   /*  guaranteed to be 0  */
```

For COFF, this corresponds to an entry in the .econst section:

```
   .sect   .econst
_zero
   .word   0
```

For EABI, this corresponds to an entry in the .const section:

```
   .sect   .const
zero
   .word   0
```

This feature is particularly useful for declaring a large table of constants, because neither time nor space is wasted at system startup to initialize the table. Additionally, the linker can be used to place the .econst or .const section in ROM.

You can use the DATA_SECTION pragma to put the variable in a section other than .econst or .const. For example, the following C code:

```
#pragma DATA_SECTION (var, ".mysect")
   const int zero=0;
```

is compiled into this assembly code (in COFF mode):

```
   .sect   .mysect
_zero
   .word   0
```

6.14 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 6.14.1.)
- ANSI/ISO C11 (--c11, see Section 6.14.2)
- ISO C++03 (--c++03 , used for all C++ files , see Section 6.2.)

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 6.14.3.)

6.14.1 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: The math library used by the compiler in C99 mode provides full C99 math support, including long double (64-bit) and float (32-bit) versions of floating point math routines. See the list of standard math.h C99 routines.
- 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

6.14.2 C11 Support (--c11)

The compiler supports the 2011 standard of C as standardized by the ISO. However, in addition to the list in Section 6.14.1, the following run-time functions and features are *not* implemented or fully supported in C11 mode:

- threads.h
- atomic operations

6.14.3 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 6.15 may conflict with strict ANSI/ISO standards, but many do not conflict with the standards.

6.15 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 6.15.6)
- __has_attribute

6.15.1 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 6-3, which is based on the list of extensions found at the GNU web site. The shaded rows describe extensions that are not supported.

Table 6-3. 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
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 numbers
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

Table 6-3. GCC Language Extensions (continued)

Extensions	Descriptions
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 6.15.6)
Vector extensions	Using vector instructions through built-in functions
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

6.15.2 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
- warn_unused_result
- weak (EABI only)

The following additional TI-specific function attributes are supported:

- retain
- ramfunc

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 has the same effect as the CODE_ALIGN pragma. See Section 6.9.3

The **always_inline** function attribute has the same effect as the FUNC_ALWAYS_INLINE pragma. See Section 6.9.11

The **calls** attribute has the same effect as the CALLS pragma, which is described in Section 6.9.1.

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 6.9.15 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 that are written as embedded assembly functions using `__asm` statements. The compiler does not generate prologue and epilog sequences for such functions. See Section 6.8.

The **noinline** function attribute has the same effect as the FUNC_CANNOT_INLINE pragma. See Section 6.9.12

The **ramfunc** attribute specifies that a function will be placed in and executed from RAM. The ramfunc attribute allows the compiler to optimize functions for RAM execution, as well as to automatically copy functions to RAM on flash-based devices. For example:

```
__attribute__((ramfunc))
void f(void) {
   ...
}
```

The --ramfunc=on option specifies that all functions compiled with this option are placed in and executed from RAM, even if this function attribute is not used.

Newer TI linker command files support the ramfunc attribute automatically by placing functions with this attribute in the .TI.ramfunc section. If you have a linker command file that does not include a section specification for the .TI.ramfunc section, you can modify the linker command file to place this section in RAM. See the *TMS320C28x Assembly Language Tools User's Guide* for details on section placement.

Fast branch instructions are generated for ramfunc functions. Regular branch instructions are generated for all other functions.

The ramfunc attribute is ignored by the CLA compiler.

The **retain** attribute has the same effect as the RETAIN pragma (Section 6.9.22). 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 6.9.4

The **weak** attribute has the same effect as the WEAK pragma (Section 6.9.25).

6.15.3 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::unroll

See Section 6.9.17.1 for an example that uses a for loop attribute.

6.15.4 Variable Attributes

The following variable attributes are supported:

- aligned
- blocked and noblocked
- deprecated
- location
- mode
- noinit (EABI only)
- persistent (EABI only)
- preserve
- retain
- section
- transparent_union
- unused
- update
- used
- weak (EABI only)

The **aligned** attribute when used on a variable has the same effect as the DATA_ALIGN pragma. See Section 6.9.5

The **blocked** and **noblocked** attributes can be used to control blocking on specific variables, including arrays and structs. See Section 3.11 for reasons to block or unblock variables. These attributes must be used on both the definition and declaration of the variable. It is recommend that these attributes be used in the header file where a variable is declared. For example:

```
__attribute__((blocked))
extern int my_array[];
__attribute__((noblocked))
extern struct_type my_struct;
```

The noblocked attribute can also be used to reference data defined in assembly or in a CLA translation unit that is not blocked.

The linker provides a diagnostic message if data access that treats a variable as blocked is attempted on non-blocked data.

The **location** attribute has the same effect as the LOCATION pragma. See Section 6.9.16. For example:

```
__attribute__((location(0x100))) extern struct PERIPH peripheral;
```

If the address specified with the location attribute is in the lower 16 bits of memory, more efficient instructions with fewer DP loads are used.

Alternately, syntax that specifies an address when dereferencing a literal can be used to allow the compiler to select more efficient instructions for handing addresses in the lower 16 bits of memory. For example, either of the following could be used:

```
#define peripheral (*((struct PERIPH)(0x100)))

struct EPWM_REGS volatile* const epwm1 = (struct EPWM_REGS *)(0x4000);
```

The **noinit** and **persistent** attributes (for EABI only) 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 *TMS320C28x 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 6.9.18.

The **preserve** and **update** attributes are used with the Live Firmware Update (LFU) capability, which supports the ability to perform a "warm start" to upgrade firmware without taking a system offline. Both attributes use the symbol address list obtained from the reference ELF image that was specified when the executable was compiled and linked. See Section 2.15 for more about LFU, which is supported for EABI only.

- The **preserve** attribute causes a global or static variable to retain its address and value from prior to the warm start. The address of this symbol remains the same as the address in the reference ELF image. Each such symbol has a `.TI.bound` section. If the .TI.bound sections are contiguous in memory, the linker can coalesce them into a single output section, which reduces the number of CINIT records required to initialize them. The following examples use the preserve attribute:

  ```
  int __attribute__((preserve))        gvar_bss_preserve;
  int __attribute__((preserve))        gvar_0 = 0x30;
  static int __attribute__((preserve)) svar_0 = 0x50;
  ```

- The **update** attribute causes a global or static variable to be re-initialized during a warm start. The address of such variables may change compared to the address in the reference ELF image. Such symbols are collected by the linker into a single `.TI.update` output section. This section defaults to copy compression (that is, no decompression is required during a warm start), which reduces the LFU image switchover time. The following examples use the update attribute:

  ```
  int __attribute__((update))        gvar_bss_update;
  int __attribute__((update))        gvar_1 = 0x50;
  static int __attribute__((update)) svar_1 = 0x70;
  ```

The **retain** attribute has the same effect as the RETAIN pragma (Section 6.9.22). 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 6.9.6

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).

The **weak** attribute has the same effect as the WEAK pragma (Section 6.9.25).

6.15.5 Type Attributes

The following type attributes are supported:

- aligned
- byte_peripheral
- deprecated
- transparent_union
- unused

You cannot create arrays using the **aligned** type attribute with an alignment larger than the size of the type. This is because the C language guarantees that the size of an array is the same as the size of each element times the number of elements. This might not be the case if an array used the aligned type attribute, because padding could be added to the array.

See Section 6.15.7 for details about the **byte_peripheral** type attribute.

6.15.6 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_sqrt()
- __builtin_sqrtf()
- __builtin_memcpy()
- __builtin_return_address()

The __builtin_frame_address() function always returns zero .

The __builtin_sqrt() and __builtin_sqrtf() functions are supported only if hardware float support is enabled. That is, __builtin_sqrt() is supported only if the Trigonometric Math Unit (TMU) is enabled. And, __builtin_sqrtf() is supported only if the TMU is enabled or using the CLA compiler.

When calling built-in functions that may be unavailable at run-time, use the Clang __has_builtin macro as shown in the following example to make sure the function is supported:

```
#if __has_builtin(__builtin_sqrt)
double estimate = __builtin_sqrt(x);
#else
double estimate = fast_approximate_sqrt(x);
#endif
```

If the built-in function is supported and the device has the appropriate hardware support, the built-in function will invoke the hardware support.

If the built-in function is supported but the device does not have the appropriate hardware enabled, the built-in function will usually become a call to an RTS library function. For example, __builtin_sqrt() will become a call to the library function sqrt().

The __builtin_return_address() function always returns zero.

6.15.7 Using the Byte Peripheral Type Attribute

The C2000 architecture has 16-bit words. Some peripherals, however, are 8-bit byte addressable. The byte peripherals bridge translates addresses between the CPU and byte peripherals by treating an address as a byte address. Therefore, only some C2000 addresses map correctly to the byte peripherals. Even and odd addresses to 16-bit data both map to the same data element on the byte peripheral.

The same is true for addresses to 32-bit data. Addresses for 16-bit accesses must be 32-bit aligned and those for 32-bit accesses must be 64-bit aligned.

C2000 driver libraries and bitfield header files are provided to access peripherals. To support correct accesses to byte peripheral data, the compiler provides the __byte_peripheral_32 intrinsic and the byte_peripheral type attribute.

The C2000 driver library accesses byte peripheral data at the correct starting address. However, on the C2000, operations on 32-bit data are often broken up into two operations on 16-bit data because these are more efficient on the architecture. Accesses to 32-bit byte peripheral data cannot be broken up regularly into two 16-bit accesses because the starting offset for the second 16-bits will be incorrect. The __byte_peripheral_32 intrinsic can be used to access a 32-bit byte peripheral data address, preventing these accesses from being broken up. The intrinsic returns a reference to an unsigned long and can be used both to read and write data. See Section 7.6 for information about this intrinsic.

The byte_peripheral type attribute can be applied as follows to typedefs of unsigned ints and unsigned longs for bitfield support.

```
typedef unsigned int bp_16 __attribute__((byte_peripheral)) ;
typedef unsigned long bp_32 __attribute__((byte_peripheral));
```

The typedef names are not significant. The attributes automatically apply the volatile keyword and handle alignment. All struct members in byte peripheral structs, whether they are bitfields or not, must have these attributes applied via typedefs to ensure proper alignment of the struct accesses. Note that struct layout is different due to differences in alignment, so the bitfields cannot always be accessed via the same container types as in regular structs.

For example, the bit positions for byte peripheral bitfield types are compared in the following examples to the positions of regular bitfields. Because 16-bit accesses must be 32-bit aligned, it is not possible to access bits at offsets 16-31 with a 16-bit container. To access these bits, a 32-bit container must be used. In example 1, you could create the same layout as in the regular case by changing the field types to bp_32 for field4 through field6. In example 2, you would change the field types to bp_32 for field2 through field4.

```
struct example1 {       // regular bits position   //byte peripherals
    bp_16 field1:9;     // 0-8                     // 0-8
    bp_16 field2:6;     // 9-14                    // 9-14
    bp_32 field3:4;     // 15-18                   // 15-18
    bp_16 field4:1;     // 19                      // 32
    bp_16 field5:5;     // 20-24                   // 33-37
    bp_16 field6:7;     // 25-31                   // 38-44
};
struct example2{        // regular bits position   //byte peripherals
    bp_32 field1:29;    // 0-28                    // 0-28
    bp_16 field2:1;     // 29                      // 32
    bp_16 field3:1;     // 30                      // 33
    bp_16 field4:1;     // 31                      // 34
};
```

Because the alignment will create padding in any objects that are declared, it is recommended that you cast byte peripheral addresses as byte peripheral struct types instead of declaring objects of those struct types.

You cannot create arrays using the byte peripheral type attribute. This is because the C language guarantees that the size of an array is the same as the size of each element times the number of elements. This would not be the case in an array of byte peripherals, because there would need to be padding in the array.

6.16 Compiler Limits

Due to the variety of host systems supported by the C/C++ compiler and the limitations of some of these systems, the compiler may not be able to successfully compile source files that are excessively large or complex. In general, exceeding such a system limit prevents continued compilation, so the compiler aborts immediately after printing the error message. Simplify the program to avoid exceeding a system limit.

Some systems do not allow filenames longer than 500 characters. Make sure your filenames are shorter than 500.

The compiler has no arbitrary limits but is limited by the amount of memory available on the host system. On smaller host systems such as PCs, the optimizer may run out of memory. If this occurs, the optimizer terminates and the shell continues compiling the file with the code generator. This results in a file compiled with no optimization. The optimizer compiles one function at a time, so the most likely cause of this is a large or extremely complex function in your source module. To correct the problem, your options are:

- Don't optimize the module in question.
- Identify the function that caused the problem and break it down into smaller functions.
- Extract the function from the module and place it in a separate module that can be compiled without optimization so that the remaining functions can be optimized.

Chapter 7

Run-Time Environment

This chapter describes the TMS320C28x 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.

7.1 Memory Model

The C28x compiler treats memory as two linear blocks of program and data memory:

- Program memory contains executable code, initialization records, and switch tables.
- Data memory contains external variables, static variables, and the system stack.

Blocks of code or data generated by a C/C++ program are placed into contiguous blocks in the appropriate memory space.

Note

The Linker Defines the Memory Map

The linker, not the compiler, defines the memory map and allocates code and data into target memory. The compiler assumes nothing about the types of memory available, about any locations not available for code or data (holes), or about any locations reserved for I/O or control purposes. The compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces. For example, you can use the linker to allocate global variables into on-chip RAM or to allocate executable code into external ROM. You can allocate each block of code or data individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although you can access physical memory locations with C/C++ pointer types).

7.1.1 Sections

The compiler produces relocatable blocks of code and data called *sections*. The sections are allocated into memory in a variety of ways to conform to a variety of system configurations. For more information about sections and allocating them, see the introductory object file information in the *TMS320C28x 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 space for the command-line arguments. See the --arg_size option for details.
  - The **.binit section** contains boot time copy tables. For details on BINIT, see the *TMS320C28x Assembly Language Tools User's Guide*.
  - The **.cinit section** contains tables for initializing variables and constants. The C28x .cinit record is limited to 16 bits. This limits initialized objects to 64K. (For EABI, the linker creates the .cinit section. For COFF, the compiler creates .cinit sections.)
  - The **.ovly section** contains copy tables for unions in which different sections have the same run address.
  - The **.init_array section** contains global constructor tables. (EABI only)
  - The **.pinit section** contains global constructor tables. (COFF only)
  - The **.c28xabi.exidx section** contains the index table for exception handling. The **.c28xabi.extab section** contains stack unwinding instructions for exception handling. See the --exceptions option for details. (EABI only)
  - 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++ qualifier *const* (provided the constant is not also defined as *volatile* or one of the exceptions described in Section 6.5.1). String literals are placed in the .const:.string subsection to enable greater link-time placement control. (EABI only)
  - The **.econst section** contains string constants, string literals, switch tables, the declaration and initialization of global and static variables, and data defined with the C/C++ qualifier *const* (provided the constant is not also defined as *volatile* or one of the exceptions described in Section 6.5.1). String literals are placed in the .econst:.string subsection to enable greater link-time placement control. (COFF only)

  - The **.data section** reserves space for non-const, initialized global and static variables. (For EABI, the compiler generates this section and it is used for initialized global and static variables. For COFF, this section may be used by assembly code, but is not used otherwise.) This section is writable.
  - The **.switch** section contains tables for switch statements. This section is placed in data memory by default. If the --unified_memory option is used, this section is placed in program memory.
  - 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.
  - The **.TI.bound section** is used to associate a symbol to a specific memory address. When either the LOCATION pragma (see Section 6.9.16) or the "location" or "preserve" variable attribute (see Section 6.15.4) is used to associate a symbol with a specific memory address, a .TI.bound section is created for the symbol. In the case of "preserve" symbols, when .TI.bound sections are contiguous in memory, the linker can coalesce them into a single output section, which reduces the number of CINIT records required to initialize them. This section may be writable or read-only.
  - The **.TI.update section** contains symbols that need to be reinitialized when a warm start occurs. Reinitialization is performed by the __TI_auto_init_warm() RTS function. It is recommended that you add an entry to the linker command file to place the .TI.update section in an appropriate memory range. This section is writable.
- **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 instead of being placed in .bss so that they can be excluded from the resulting application. (EABI only for compiler; EABI and COFF for assembler)
  - The **.ebss section** reserves space for global and static variables defined. At program startup time, the C/C++ boot routine copies data out of the .cinit section (which can be in ROM) and uses it for initializing variables in the .ebss section. (COFF only)
  - The **.stack section** reserves memory for the C/C++ software stack. This memory is used to pass arguments to functions and to allocate space for local variables.
  - The **.esysmem 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 .esysmem section. (COFF only)
  - The **.sysmem section** reserves space for dynamic memory allocation. The reserved 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. (EABI only)

The assembler creates the default sections .text, .ebss or .bss (depending on the ABI), and .data. You can instruct the compiler to create additional sections by using the CODE_SECTION and DATA_SECTION pragmas (see Section 6.9.4 and Section 6.9.6).

The linker takes the individual sections from different object files and combines sections that have the same name. The resulting output sections and the appropriate placement in memory for each section are listed in Table 7-1. You can place these output sections anywhere in the address space as needed to meet system requirements.

The linker also creates some additional sections not referenced by the compiler. For example, the **.common** section contains common-block symbols allocated by the linker.

For EABI, it is encouraged that you use a unified memory scheme that places all sections on page 0. However, the default linker command file specifies page 0 or 1 for sections as indicated in the following table. In general, uninitialized and constant value sections are placed on page 1 by the linker command file; all other sections are generally placed on page 0.

Table 7-1. Summary of Sections and Memory Placement

Section	Type of Memory	Default Page
.binit	expected to be in FLASH/ROM	1
.bss (EABI only)	must be in RAM	1
.ebss (COFF only)	must be in RAM	1
.c28xabi.exidx (EABI only)	expected to be in FLASH/ROM	1
.c28xabi.extab (EABI only)	expected to be in FLASH/ROM	1
.cinit (1)	expected to be in FLASH/ROM	0
.const (EABI only)	expected to be in FLASH/ROM	1
.econst (COFF only)	expected to be in FLASH/ROM	1
.data (used mainly by EABI)	must be in RAM	0
.init_array (EABI only)	expected to be in FLASH/ROM	0
.pinit (COFF only)	expected to be in FLASH/ROM	0
.ppdata	must be in RAM	1
.stack	must be in RAM	1
.switch	depends on the --unified_memory option setting	0, 1
.sysmem (EABI only)	must be in RAM	1
.esysmem (COFF only)	must be in RAM	1
.text	expected to be in FLASH/ROM	0

(1) The .cinit section is created by the compiler for COFF and by the linker for EABI.

You can use the SECTIONS directive in the linker command file to customize the section-allocation process. For more information about allocating sections into memory, see the linker description chapter in the *TMS320C28x Assembly Language Tools User's Guide*.

7.1.2 C/C++ System Stack

The C/C++ compiler uses a stack to:

- Allocate local variables
- Pass arguments to functions
- Save the processor status
- Save function return addresses
- Save temporary results

The run-time stack grows up from low addresses to higher addresses. By default, the stack is allocated in the .stack section. (See the run-time-support boot.asm file.) The compiler uses the hardware stack pointer (SP) to manage this stack.

Note

Linking the .stack Section: The .stack section has to be linked into the low 64K of data memory. The SP is a 16-bit register and cannot access addresses beyond 64K.

For frames that exceed 63 words in size (the maximum reach of the SP offset addressing mode), the compiler uses XAR2 as a frame pointer (FP). Each function invocation creates a new frame at the top of the stack, from which local and temporary variables are allocated. The FP points at the beginning of this frame to access memory locations that cannot be referenced directly using the SP.

The stack size is set by the linker. The linker also creates a global symbol, __STACK_SIZE (for COFF) or __TI_STACK_SIZE (for EABI), and assigns it a value equal to the size of the stack in bytes. The default stack size is 1K words. You can change the size of the stack at link time by using the --stack_size linker option.

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 2.14.

7.1.3 Allocating .econst to Program Memory

Note

This section applies to applications that use the COFF ABI.

If your system configuration does not support allocating an initialized section such as .econst to data memory, then you have to allocate the .econst section to load in program memory and run in data memory. At boot time, copy the .econst section from program to data memory. The following sequence shows how you can perform this task.

1. Extract boot.asm from the source library:

```
ar2000 -x rts.src boot.asm
```

2. Edit boot.asm and change the CONST_COPY flag to 1:

```
CONST_COPY  .set 1
```

3. Assemble boot.asm:

```
cl2000 boot.asm
```

4. Archive the boot routine into the object library:

```
ar2000 -r rts2800_ml.lib boot.c.obj
```

For an .econst section, link with a linker command file that contains the following entries:

```
SECTIONS
{  ...
   .econst : load = PROG PAGE 1, run = DATA PAGE 1
             {
                /* GET RUN ADDRESS */
                __econst_run = .;
                /* MARK LOAD ADDRESS */
                *(.ec_mark)
                /* ALLOCATE .econst */
                *(.econst)
                /* COMPUTE LENGTH */
                __econst_length = - .__econst_run;
             }
}
```

In your linker command file, you can substitute the name PROG with the name of a memory area on page 0 and DATA with the name of a memory area on page 1. The rest of the command file must use the names as above. The code in boot.asm that is enabled when you change CONST_COPY to 1 depends on the linker command file using these names in this manner. To change any of the names, you must edit boot.asm and change the names in the same way.

7.1.4 Dynamic Memory Allocation

The run-time-support library supplied with the C28x 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 .esysmem or .sysmem section. You can set the size of the .esysmem or .sysmem section by using the --heap_size=*size* option with the linker command. The linker also creates a global symbol, __SYSMEM_SIZE (for COFF) or __TI_SYSMEM_SIZE (for EABI),, and assigns it a value equal to the size of the heap in words. The default size is 1K words. For more information on the --heap_size option, see the linker description chapter in the *TMS320C28x 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 (.esysmem or .sysmem); therefore, the dynamic memory pool can have a size limited only by the amount of available memory in your system. To conserve space in the .ebss or .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. For example, allocating a variable-length array requires at least 1500 words of heap memory.

7.1.5 Initialization of Variables

The C/C++ compiler produces code that is suitable for use as firmware in a ROM-based system. In such a system, the initialization tables in the .cinit section are stored in ROM. At system initialization time, the C/C++ boot routine copies data from these tables (in ROM) to the initialized variables in .ebss or .bss (RAM).

When a program is loaded directly from an object file into memory and run, you can avoid having the .cinit section occupy space in memory. A loader can read the initialization tables directly from the object file (instead of from ROM) and perform the initialization directly at load time instead of at run time. You can specify this to the linker by using the --ram_model link option. For more information, see Section 7.10.

7.1.6 Allocating Memory for Static and Global Variables

A unique space is allocated for all static variables declared in a C/C++ program. The linker determines the address of the space. The compiler ensures that space for these variables is allocated in multiples of words so that each variable is aligned on a word boundary.

The compiler expects global variables to be allocated into data memory. (For COFF it reserves space for such variables in .ebss. For EABI, it reserves space in .data or in sections named with the DATA_SECTION pragma, which is described in Section 6.9.6.) Variables declared in the same module are allocated into a single block of memory. The block is likely to be contiguous, but this is not guaranteed.

7.1.7 Field/Structure Alignment

When the compiler determines the layout of a structure, it provides as many words as are needed to hold all of the members, while complying with each member's alignment constraint. This means that padding bytes may be placed between members and at the end of the structure. Each member is aligned as its type requires.

Types with a size of 16 bits are aligned on 16-bit boundaries. Types with a size of 32 bits or larger are aligned on 32-bit (2 word) boundaries. For details on EABI field alignment, refer to the *C28x Embedded Application Binary Interface (EABI) Reference Guide* (SPRAC71).

However, bit-fields may not be aligned as strictly as the declared type of the bit-field would be if it were not a bit-field. Bit-fields are packed in the order seen in the source code. The least significant bits of the structure word

are filled first. Bit-fields are allocated only as many bits as requested. Bit-fields are packed into adjacent bits of a word.

For COFF, bit-fields do not overlap word boundaries. If a bit-field would overlap into the next word, the entire bit-field is placed into the next word.

7.1.8 Character String Constants

In C, a character string constant is used in one of the following ways:

- To initialize an array of characters. For example:

```
char s[] = "abc";
```

  When a string is used as an initializer, it is simply treated as an initialized array; each character is a separate initializer. For more information about initialization, see Section 7.10.
- In an expression. For example:

```
strcpy (s, "abc");
```

  When a string is used in an expression, the string itself is defined in the .econst or .const (depending on the ABI) section with the .string assembler directive, along with a unique label that points to the string; the terminating 0 byte is included. For example, the following lines define the string abc, and the terminating 0 byte (the label SL5 points to the string):

```
      .sect   ".econst"
SL5: .string "abc",0
```

  String labels have the form SL*n*, where *n* is a number assigned by the compiler to make the label unique. The number begins at 0 and is increased by 1 for each string defined. All strings used in a source module are defined at the end of the compiled assembly language module.

  The label SL*n* represents the address of the string constant. The compiler uses this label to reference the string expression.

  Because strings are stored in the .econst or .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 */
```

7.2 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. There are two types of register variable registers, save on entry and save on call. The distinction between these two types of registers is the method by which they are preserved across function calls. It is the called function's responsibility to preserve save-on-entry registers, and the calling function's responsibility to preserve save-on-call registers if you need to preserve that register's value.

7.2.1 TMS320C28x Register Use and Preservation

Table 7-2 summarizes how the compiler uses the TMS320C28x registers and shows which registers are defined to be preserved across function calls. The FPU uses all the C28x registers as well as the registers described in Table 7-3.

See the *TMS320C28x CPU and Instruction Set Reference Guide* (SPRU430) for details about C28x registers.

Table 7-2. Register Use and Preservation Conventions

Register	Usage	Save on Entry	Save on Call
AL	Expressions, argument passing, and returns 16-bit results from functions	No	Yes
AH	Expressions and argument passing	No	Yes
DP	Data page pointer (used to access global variables)	No	No
PH	Multiply expressions and Temp variables	No	Yes
PL	Multiply expressions and Temp variables	No	Yes
SP	Stack pointer	(1)	(1)
T	Multiply and shift expressions	No	Yes
TL	Multiply and shift expressions	No	Yes
XAR0	Pointers and expressions	No	Yes
XAR1	Pointers and expressions	Yes	No
XAR2	Pointers, expressions, and frame pointing (when needed)	Yes	No
XAR3	Pointers and expressions	Yes	No
XAR4	Pointers, expressions, argument passing, and returns 32-bit pointer values from functions	No	Yes
XAR5	Pointers, expressions, and arguments	No	Yes
XAR6	Pointers and expressions	No	Yes
XAR7	Pointers, expressions, indirect calls and branches (used to implement pointers to functions and switch statements)	No	Yes

(1) The SP is preserved by the convention that everything pushed on the stack is popped off before returning.

Table 7-3. FPU Register Use and Preservation Conventions

FPU32 Registers	FPU64 Registers	Usage	Save on Entry	Save on Call
R0H	R0 = R0H:R0L	Expressions, argument passing, and returns 32-bit float from functions	No	Yes
R1H	R1 = R1H:R1L	Expressions and argument passing	No	Yes
R2H	R2 = R2H:R2L	Expressions and argument passing	No	Yes
R3H	R3 = R3H:R3L	Expressions and argument passing	No	Yes
R4H	R4 = R4H:R4L	Expressions	Yes	No
R5H	R5 = R5H:R5L	Expressions	Yes	No
R6H	R6 = R6H:R6L	Expressions	Yes	No
R7H	R7 = R7H:R7L	Expressions	Yes	No

7.2.2 Status Registers

Table 7-4 shows all of the status fields used by the compiler. Presumed value is the value the compiler expects in that field upon entry to, or return from, a function; a dash in this column indicates the compiler does not expect a particular value. The modified column indicates whether code generated by the compiler ever modifies this field. See the *TMS320C28x CPU and Instruction Set Reference Guide* (SPRU430) for details about status registers.

Table 7-4. Status Register Fields

Field	Name	Presumed Value	Modified
ARP	Auxiliary Register Pointer	-	Yes
C	Carry	-	Yes
N	Negative flag	-	Yes
OVM	Overflow mode	0(1)	Yes
PAGE0	Direct/stack address mode	0(1)	No
PM	Product shift mode	1(1) (2)	Yes
SPA	Stack pointer align bit	-	Yes (in interrupts)
SXM	Sign extension mode	-	Yes
TC	Test/control flag	-	Yes
V	Overflow flag	-	Yes
Z	Zero flag	-	Yes

(1) The initialization routine that sets up the C run-time environment sets these fields to the presumed value.
(2) The result is not shifted after a multiply if PM=1. If PM=0, the result is left shifted by 1 after each multiply.

Table 7-5 shows the additional status fields used by the compiler for FPU targets. See the *TMS320C28x Extended Instruction Sets Technical Reference Manual* (SPRUHS1) for details about these registers.

Table 7-5. Floating-Point Status Register (STF(1)) Fields For FPU Targets Only

Field	Name	Presumed Value	Modified
LVF(2) (3)	Latched overflow float flag	-	Yes
LUF(2) (3)	Latched underflow float flag	-	Yes
NF(2)	Negative float flag	-	Yes
ZF(2)	Zero float flag	-	Yes
NI(2)	Negative integer flag	-	Yes
ZI(2)	Zero integer flag bit	-	Yes
TF(2)	Test flag bit	-	Yes
RNDF32	Round F32 mode(4)	-	Yes
RNDF64	Round F64 mode(4)	-	Yes
SHDWS	Shadow mode status	-	Yes

(1) Unused STF register bits read 0 and writes are ignored.
(2) Specified flags in the STF register can be exported to the ST0 register by way of the MOVST0 instruction.
(3) The LVF and LUF flag signals can be connected to the PIE to generate an overflow-and-underflow interrupt. This can be a useful debug tool.
(4) If RNDF32 or RNDF64 is 0, mode is round to zero (truncate), otherwise mode is round to nearest (even).

All other status register fields are not used and do not affect code generated by the compiler.

7.3 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 *C28x Embedded Application Binary Interface (EABI) Reference Guide* (SPRAC71).

Figure 7-1 illustrates a typical function call. In this example, parameters that cannot be placed in registers are passed to the function on the stack. The function then allocates local variables and calls another function. This example shows the allocated local frame and argument block for the called function. Functions that have no local variables and do not require an argument block do not allocate a local frame.

The term *argument block* refers to the part of the local frame used to pass arguments to other functions. Parameters are passed to a function by moving them into the argument block rather than pushing them on the stack. The local frame and argument block are allocated at the same time.

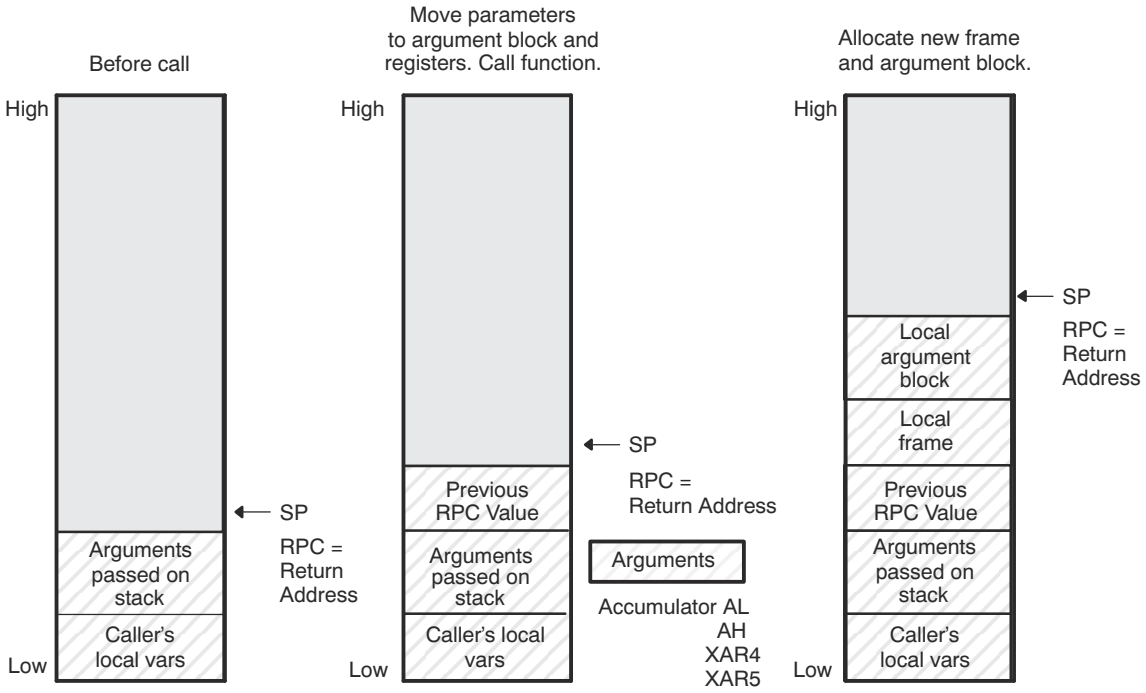


Figure 7-1. Use of the Stack During a Function Call

7.3.1 How a Function Makes a Call

A function (parent function) performs the following tasks when it calls another function (child function).

```
struct big { long x[10]; };
 struct small { int x; };
```

T0 T0 AC0 AR0

```
int fn(int i1, long l2, int *p3);
```

AC0 AR0 T0 T1 AR1

```
long fn(int *p1, int i2, int i3, int i4);
```

AR0 AR1

```
struct big fn(int *p1);
```

T0 AR0 AR1

```
int fn(struct big b, int *p1);
```

AC0 AR0

```
struct small fn(int *p1);
```

T0 AC0 AR0

```
int fn(struct small b, int *p1);
```

T0 stack stack...

```
int printf(char *fmt, ...);
```

AC0 AC2 AC2 stack T0

```
void fn(long l1, long l2, long l3, long l4, int i5);
```

AC0 AC1 AC2 AR0 AR1

```
void fn(long l1, long l2, long l3, int *p4, int *p5,
```

AR2 AR3 AR4 T0 T1

```
int *p6, int *p7, int *p8, int i9, int i10);
```

1. Any registers whose values are not necessarily preserved by the function being called (registers that are not save-on-entry (SOE) registers), but will be needed after the function returns are saved on the stack.
2. If the called function returns a structure, the calling function allocates the space for the structure and pass the address of that space to the called function as the first argument.

3. Arguments passed to the called function are placed in registers and, when necessary, placed on the stack. Arguments are placed in registers using the following scheme:
   a. If the target is FPU and there are any 32-bit float arguments, the first four float arguments are placed in registers R0H-R3H.
   b. If there are any 64-bit floating point arguments (long doubles), they are passed by reference.
   c. If there are any 64-bit integer arguments (long long), the first is placed in ACC and P (ACC holds the upper 32 bits and P holds the lower 32 bits). All other 64-bit integer arguments are placed on the stack in reverse order.

      If the P register is used for argument passing, then prolog/epilog abstraction is disabled for that function. See Section 3.13 for more information on abstraction.
   d. If there are any 32-bit arguments (longs or floats), the first is placed in the 32-bit ACC (AH/AL). All other 32-bit arguments are placed on the stack in reverse order.

```
func1(long a, long long b, int c, int* d);
      stack   ACC/P   XAR5,  XAR4
```

   e. Pointer arguments are placed in XAR4 and XAR5. All other pointers are placed on the stack.
   f. Remaining 16-bit arguments are placed in the order AL, AH, XAR4, XAR5 if they are available.
4. Any remaining arguments not placed in registers are pushed on the stack in reverse order. That is, the leftmost argument that is placed on the stack is pushed on the stack last. All 32-bit arguments are aligned to even addresses on the stack.

   A structure argument is passed as the address of the structure. The called function must 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.
5. The stack pointer (SP) must be even-aligned by the parent function prior to making a call to the child function. This is done by incrementing the stack pointer by 1, if necessary. If needed, the coder should increment the SP before making the call. These function call examples show where arguments are placed:

```
func1 (int a, int b. long c)
       XAR4   XAR5    AH/AL
func1 (long a, int b, long c) ;
       AH/AL   XAR4   stack
vararg (int a, int b, int c, ...)
        AL     AH     stack
```

6. The caller calls the function using the LCR instruction. The RPC register value is pushed on the stack. The return address is then stored in the RPC register.
7. The stack is aligned at function boundary.

7.3.2 How a Called Function Responds

A called function (child function) must perform the following tasks:

1. If the called function modifies XAR1, XAR2, or XAR3, it must save them, since the calling function assumes that the values of these registers are preserved upon return. If the target is FPU, then in addition to the C28x registers, the called function must save registers R4H, R5H, R6H or R7H, if it modifies any of them. Any other registers may be modified without preserving them.
2. The called function allocates enough space on the stack for any local variables, temporary storage area, and arguments to functions that this function might call. This allocation occurs once at the beginning of the function by adding a constant to the SP register.
3. The stack is aligned at function boundary.
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 passes 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 properly declare functions 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. The called function returns a value. It is placed in a register using the following convention:

16-bit integer value	AL
32-bit integer value	ACC
64-bit integer value	ACC/P
32-bit pointer	XAR4
structure reference	XAR6

If the target is FPU and a 32-bit float value is returned, the called function places this value in R0H.

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 XAR6. To return a structure, the called function copies the structure to the memory block pointed 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 properly declare functions 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). Returning 64-bit floating-point values (long double) are returned similarly to structures.

7. The called function deallocates the frame by subtracting the value that was added to the SP earlier.
8. The called function restores the values of all registers saved in Step 1.
9. The called function returns using the LRETR instruction. The PC is set to the value in the RPC register. The previous RPC value is popped from the stack and stored in the RPC register.

7.3.3 Special Case for a Called Function (Large Frames)

If the space that needs to be allocated on the stack (step 2 in the previous section) is larger than 63 words, additional steps and resources are required to ensure that all local nonregister variables can be accessed. Large frames require using a frame pointer register (XAR2) to reference local non-register variables within the frame. Prior to allocating space on the frame, the frame pointer is set up to point at the first argument on the stack that was passed on to the called function. If no incoming arguments are passed on to the stack, the frame pointer points to the return address of the calling function, which is at the top of the stack upon entry to the called function.

Avoid allocating large amounts of local data when possible. For example, do not declare large arrays within functions.

7.3.4 Accessing Arguments and Local Variables

A function accesses its local nonregister variables and its stack arguments indirectly through either the SP or the FP (frame pointer, designated to be XAR2). All local and argument data that can be accessed with the SP use the *−SP [offset] addressing mode since the SP always points one past the top of the stack and the stack grows toward larger addresses.

Note

The PAGE0 Mode Bit Must Be Reset: Since the compiler uses the *-SP [offset] addressing mode, the PAGE0 mode bit must be reset (set to 0).

The largest offset available using *-SP [offset] is 63. If an object is too far away from the SP to use this mode of access, the compiler uses the FP (XAR2). Since FP points at the bottom of the frame, accesses made with the FP use either *+FP [offset] or *+FP [AR0/AR1] addressing modes. Since large frames require utilizing XAR2 and possibly an index register, extra code and resources are required to make local accesses.

7.3.5 Allocating the Frame and Accessing 32-Bit Values in Memory

Some TMS320C28x instructions read and write 32 bits of memory at once (MOVL, ADDL, etc.). These instructions require that 32-bit objects be allocated on an even boundary. To ensure that this occurs, the compiler takes these steps:

1. It initializes the SP to an even boundary.
2. Because a call instruction adds 2 to the SP, it assumes that the SP is pointing at an even address.
3. It makes sure that the space allocated on the frame totals an even number, so that the SP points to an even address.
4. It makes sure that 32-bit objects are allocated to even addresses, relative to the known even address in the SP.
5. Because interrupts cannot assume that the SP is odd or even, it aligns the SP to an even address.

For more information on how these instructions access memory, see the *TMS320C28x Assembly Language Tools User's Guide*.

7.4 Accessing Linker Symbols in C and C++

See the section on "Using Linker Symbols in C/C++ Applications" in the *TMS320C28x Assembly Language Tools User's Guide* for information about referring to linker symbols in C/C++ code.

7.5 Interfacing C and C++ With Assembly Language

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

- Use separate modules of assembled code and link them with compiled C/C++ modules (see Section 7.5.1).
- Use assembly language variables and constants in C/C++ source (see Section 7.5.3).
- Use inline assembly language embedded directly in the C/C++ source (see Section 7.5.5).
- Use intrinsics in C/C++ source to directly call an assembly language statement (see Section 7.6).

7.5.1 Using Assembly Language Modules With C/C++ Code

Interfacing C/C++ with assembly language functions is straightforward if you follow the calling conventions defined in Section 7.3, and the register conventions defined in Section 7.2. C/C++ code can access variables and call functions defined in assembly language, and assembly code can access C/C++ variables and call C/C++ functions.

Follow these guidelines to interface assembly language and C:

- All functions, whether they are written in C/C++ or assembly language, must follow the register conventions outlined in Section 7.2.
- Dedicated registers modified by a function must be preserved. Dedicated registers include:

XAR1	R4H (FPU only)
XAR2	R5H (FPU only)
XAR3	R6H (FPU only)
SP	R7H (FPU only)

  If the SP is used normally, it does not need to be preserved explicitly. The assembly function is free to use the stack as long as anything that is pushed on the stack is popped back off before the function returns (thus preserving the SP).

  Any register that is not dedicated can be used freely without being preserved.

- The stack pointer (SP) must be even-aligned by the parent function prior to making a call to the child function. This is done by incrementing the stack pointer by 1, if necessary. If needed, the coder should increment the SP before making the call.
- The stack is aligned at function boundary.
- Interrupt routines must save *all* the registers they use. For more information, see Section 7.7.
- 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 7.3.1.

  When accessing arguments passed in from a C/C++ function, these same conventions apply.
- Longs and floats are stored in memory with the least significant word at the lower address.
- Structures are returned as described in Section 7.3.2.
- 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 prepends an underscore (_) to the beginning of all identifiers. In assembly language modules, you must use the prefix _ for all objects that are to be accessible from C/C++. For example, a C/C++ object named x is called _x in assembly language. For identifiers that are to be used only in an assembly language module or modules, any name that does not begin with an underscore can be safely used without conflicting with a C/C++ identifier.
- 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 6.12 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.
- Because compiled code runs with the PAGE0 mode bit reset, if you set the PAGE0 bit to 1 in your assembly language function, you must set it back to 0 before returning to compiled code.
- If you define a structure in assembly and access it in C using extern struct, the structure should be blocked. The compiler assumes that structure definitions are blocked to optimize the DP load. So the definition should honor this assumption. You can block the structure by specifying the blocking flag in the .usect directive. See the *TMS320C28x Assembly Language Tools User's Guide* for more information on these directives.

7.5.2 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 7-1 illustrates a C++ function called main, which calls an assembly language function called asmfunc, Example 7-2. The asmfunc function takes its single argument, adds it to the C++ global variable called gvar, and returns the result.

Example 7-1. Calling an Assembly Language Function From 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 7-2. Assembly Language Program Called by Example 7-1

```
        .global _gvar
        .global _asmfunc
_asmfunc:
        MOVZ    DP,#_gvar
        ADDB    AL,#5
        MOV     @_gvar,AL
        LRETR
```

In the C++ program in Example 7-1, the extern “C” declaration tells the compiler to use C naming conventions (i.e., no name mangling). When the linker resolves the .global _asmfunc reference, the corresponding definition in the assembly file will match.

The parameter i is passed in register AL.

7.5.3 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 .ebss or .bss section, a variable not defined in the .ebss or .bss section, or a linker symbol.

7.5.3.1 Accessing Assembly Language Global Variables

Accessing variables from the .ebss or .bss section or a section named with .usect is straightforward:

1. Use the .bss or .usect directive to define the variable.
2. Use the .def or .global directive to make the definition external.
3. Use the appropriate linkname in assembly language.
4. In C/C++, declare the variable as *extern* and access it normally.

Example 7-4 and Example 7-3 show how you can access a variable defined in .ebss.

Example 7-3. Assembly Language Variable Program

```
* Note the use of underscores in the following lines
       .ebss     _var,1    ; Define the variable
       .global   _var      ; Declare it as external
```

Example 7-4. C Program to Access Assembly Language From Example 7-3

```
extern int var;         /* External variable */
var = 1;                /* Use the variable  */
```

7.5.3.2 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 *TMS320C28x 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 *TMS320C28x 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 7-5 and Example 7-6.

Example 7-5. Accessing an Assembly Language Constant From C

```
extern int table_size;        /*external ref */
#define TABLE_SIZE ((int) (&table_size))
      .                       /* use cast to hide address-of */
      .
      .
for (I=0; i<TABLE_SIZE; ++I)  /* use like normal symbol */
```

Example 7-6. Assembly Language Program for Example 7-5

```
_table_size  .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 7-5, int is used. You can reference linker-defined symbols in a similar manner.

7.5.4 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 *TMS320C28x Assembly Language Tools User's Guide*.

7.5.5 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 6.8.

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.

7.6 Using Intrinsics to Access Assembly Language Statements

The C28x 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 double underscore, and are accessed by calling them as you do a function. For example:

```
long lvar;
int  ivar;
unsigned int uivar;
lvar = __mpyxu(ivar, uivar);
```

The intrinsics listed in Table 7-6 are available. They correspond to the indicated TMS320C28x assembly language instruction(s). See the *TMS320C28x CPU and Instruction Set Reference Guide* for more information.

Table 7-6. TMS320C28x C/C++ Compiler Intrinsics

Intrinsic	Assembly Instruction(s)	Description
int **__abs16_sat(** int *src* **);**	**SETC OVM** **MOV AH,** *src* **ABS ACC** **MOV** *dst* **, AH** **CLRC OVM**	Clear the OVM status bit. Load *src* into AH. Take absolute value of ACC. Store AH into *dst*. Clear the OVM status bit.
void **__add(** int * *m* **,** int *b* **);**	**ADD** * *m* **,** *b*	Add the contents of memory location *m* to *b* and store the result in *m*, in an atomic way.
long **__addcu(** long *src1* **,** unsigned int *src2* **);**	**ADDCU ACC,** {*mem* | *reg*}	The contents of *src2* and the value of the carry bit are added to ACC. The result is in ACC.
void **__addl(** long * *m* **,** long *b* **);**	**ADDL** * *m* **,** *b*	Add the contents of memory location *m* to *b* and store the result in *m*, in an atomic way.
void **__and(** int * *m* **,** int *b* **);**	**AND** * *m* **,** *b*	AND the contents of memory location *m* to *b* and store the result in m, in an atomic way.
int &**__byte(** int * *array* **,** unsigned int *byte_index* **);**	**MOVB** *array* **[** *byte_index* **].LSB,** *src* or **MOVB** *dst* **,** *array* **[** *byte_index* **].LSB**	The lowest addressable unit in C28x is 16 bits. Therefore, normally you cannot access 8-bit entities off a memory location. This intrinsic helps access an 8-bit quantity off a memory location, and can be invoked as follows: `__byte(array,5) = 10;` `b = __byte(array,20);`
unsigned long &*y* **__byte_peripheral_32(**unsigned long **x* **);**		Used to access a 32-bit byte peripheral data address without the access being broken in half. The intrinsic returns a reference to an unsigned long and can be used both to read and write data. See Section 6.15.7.
void **__dec(** int * *m* **);**	**DEC** * *m*	Decrement the contents of memory location *m* in an atomic way.
unsigned int **__disable_interrupts( );**	**PUSH ST1** **SETC INTM, DBGM** **POP** *reg16*	Disable interrupts and return the old value of the interrupt vector.
void **__dmac(** long **src1*, long **src2*, long *&accum1*, long *&accum2*, int *shift*);	**SPM** *n* ; the PM value required for *shift* **MOVL ACC,** *accum1* **MOVL P**, *accum2* **MOVL XARx,** *src1* **MOVL XAR7,** *src2* **DMAC ACC:P, *XARx++, *XAR7++**	Set the required PM value for shift. Move accum1 and accum2 into ACC and P. Move the addresses src1 and src2 into XARx and XAR7. ACC = ACC + (src1[i+1] * src2[i+1]) << PM P = P + (src1[i] * src2[i]) << PM See Section 3.15.3 for more information.
void **__eallow(** void **);**	**EALLOW**	Permits the CPU to write freely to protected registers.

Table 7-6. TMS320C28x C/C++ Compiler Intrinsics (continued)

Intrinsic	Assembly Instruction(s)	Description
void **__edis(** void **);**	**EDIS**	Prevents the CPU from writing freely to protected registers after EALLOW is used.
unsigned int **__enable_interrupts( );**	**PUSH ST1** **CLRC INTM, DBGM** **POP** *reg16*	Enable interrupts and return the old value of the interrupt vector.
uint32_t **__f32_bits_as_u32(** float *src*);	--	Extracts the bits in a float as a 32-bit register. This intrinsic generates no code; it tells the compiler to change how it interprets the bits in place. See Section 7.6.1 for examples.
uint64_t **__f64_bits_as_u64(** double *src*);	--	Extracts the bits in a double as a 64-bit register. This intrinsic generates no code; it tells the compiler to change how it interprets the bits in place. See Section 7.6.1 for examples.
int **__flip16(**int *src* **);**	**FLIP AX**	Reverses order of bits in int *src*.
long **__flip32(**long *src* **);**	**FLIP AX**	Reverses order of bits in long *src*.
long long **__flip64(**long long *src* **);**	**FLIP AX**	Reverses order of bits in long long *src*.
void **__inc(** int * *m* **);**	**INC** * *m*	Increment the contents of memory location *m* in an atomic way.
long**=__IQ(** long double *A* **,** int *N* **);**		Convert the long double *A* into the correct IQN value returned as a long type. If both arguments are constants the compiler converts the arguments to the IQ value during compile time. Otherwise a call to the RTS routine, __IQ, is made. This intrinsic cannot be used to initialize global variables to the .cinit section.
long *dst* **=__IQmpy(** long *A* **,** long *B* **,** int *N* **);**		Perform optimized multiplication using the C28 IQmath library. The *dst* becomes ACC or P, *A* becomes XT:
If *N* == 0:	**IMPYL** {**ACC**|**P**}**, XT, B**	The dst is ACC or P. If *dst* is ACC, the instruction takes 2 cycles. If dst is P, the instruction takes 1 cycle.
If 0 < *N* < 16:	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **ASR64 ACC:P, #** *N*	
If 15 < *N* < 32:	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **LSL64 ACC:P, #(32-** *N* **)**	
If *N* == 32:	**QMPYL** {**ACC**|**P**}**, XT, B**	
If *N* is a variable:	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **MOV T,** *N* **LSR64 ACC:P, T**	
long *dst* **= __IQsat(** long *A* **,** long *max* **,** long *min* **);**		The *dst* becomes ACC. Different code is generated based on the value of *max* and/or *min*. Calling __IQsat with *max* < *min* results in undefined behavior.
If *max* and *min* are 22-bit unsigned constants:	**MOVL ACC, A** **MOVL XAR** *n* **, #** *22bits* **MINL ACC, XAR** *n* **MOVL XAR** *n* **, #** *22bits* **MAXL ACC, XAR** *n*	
If *max* and *min* are other constants:	**MOVL ACC, A** **MOV PL, #** *max lower 16 bits* **MOV PH, #** *max upper 16 bits* **MINL ACC, P** **MOV PL, #** *min lower 16 bits* **MOV PH, #** *min upper 16 bits* **MAXL ACC, P**	

Table 7-6. TMS320C28x C/C++ Compiler Intrinsics (continued)

Intrinsic	Assembly Instruction(s)	Description
If *max* and/or *min* are variables:	**MOVL ACC, A** **MINL ACC,** *max* **MAXL ACC,** *min*	
long *dst* **= __IQxmpy(**long *A* **,** long *B* **,** int *N* **);**		Perform optimized multiplication by a power of 2 using the C28 IQmath library. The *dst* becomes ACC or P; *A* becomes XT. Code is generated based on the value of *N*.
If *N* == 0:	**IMPYL ACC/P, XT, B**	The *dst* is in ACC or P.
If 0 < *N* < 17:	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **LSL64 ACC:P, #** *N*	The *dst* is in ACC.
If 0 > *N* > -17:	**QMPYL ACC, XT, B** **SETC SXM** **SFR ACC, #abs(** *N* **)**	The *dst* is in ACC.
If 16 < *N* < 32:	**IMPYL P, XT, B** **QMPYL ACC, XT, B** **ASR64 ACC:P, #** *N*	The *dst* is in P.
If *N* == 32:	**IMPYL P, XT, B**	The *dst* is in P.
If -16 > *N* > -33	**QMPYL ACC, XT, B** **SETC SXM SRF ACC, #16** **SRF ACC, #abs(** *N* **)−16**	The *dst* is in ACC.
If 32 < *N* < 49:	**IMPYL ACC, XT, B** **LSL ACC, #** *N* **-32**	The *dst* is in ACC.
If -32 > *N* > -49:	**QMPYL ACC, XT, B** **SETC SXM SFR ACC, #16** **SFR ACC, #16**	The *dst* is in ACC.
If 48 < *N* < 65:	**IMPYL ACC, XT, B** **LSL64 ACC:P, #16** **LSL64 ACC:P, #** *N* **−48**	The *dst* is in ACC.
If -48 > *N* > -65:	**QMPYL ACC, XT, B** **SETC SXM SFR ACC, #16** **SFR ACC, #16**	The *dst* is in ACC.
long long **__llmax(**long long *dst* **,** long long *src* **);**	**MAXL ACC,** *src.hi32* **MAXCUL P,** *src.lo32*	If *src* > *dst*, copy src to dst.
long long **__llmin(**long long *dst* **,** long long *src* **);**	**MINL ACC,** *src.hi32* **MINCUL P,** *src.lo32*	If *src* < *dst*, copy src to dst
long **__lmax(**long *dst* **,** long *src* **);**	**MAXL ACC,** *src*	If *src* > *dst*, copy src to dst.
long **__lmin(**long *dst* **,** long *src* **);**	**MINL ACC,** *src*	If *src* < *dst*, copy src to dst
int **__max(**int *dst* **,** int *src* **);**	**MAX** *dst* **,** *src*	If *src* > *dst*, copy src to dst
int **__min(**int *dst* **,** int *src* **);**	**MIN** *dst* **,** *src*	If *src* < *dst*, copy src to dst
int **__mov_byte(** int **src* **,** unsigned int *n* **);**	**MOVB AX.LSB,*+XARx[** *n* **]** or **MOVZ AR0/AR1, @** *n* **MOVB AX.LSB,*XARx[** {AR0|AR1} **]**	Return the 8-bit *n*th element of a byte table pointed to by *src*. This intrinsic is provided for backward compatibility. The intrinsic __byte is preferred as it returns a reference. Nothing can be done with __mov_byte() that cannot be done with __byte().
long **__mpy(** int *src1* **,** int *src2* **);**	**MPY ACC,** *src1* **, #** *src2*	Move *src1* to the T register. Multiply T by a 16-bit immediate (*src2*). The result is in ACC.
long **__mpyb(** int *src1* **,** uint *src2* **);**	**MPYB** {ACC | P}**, T, #** *src2*	Multiply *src1* (the T register) by an unsigned 8-bit immediate (*src2*). The result is in ACC or P.
long **__mpy_mov_t(** int *src1* **,** int *src2* **,** int * *dst2* **);**	**MPY ACC, T,** *src2* **MOV @** *dst2* **, T**	Multiply *src1* (the T register) by *src2*. The result is in ACC. Move *src1* to **dst2*.

Table 7-6. TMS320C28x C/C++ Compiler Intrinsics (continued)

Intrinsic	Assembly Instruction(s)	Description
unsigned long **__mpyu(**unit *src2* **,** unit *srt2* **);**	**MPYU** {ACC | P}**, T,** *src2*	Multiply *src1* (the T register) by *src2*. Both operands are treated as unsigned 16-bit numbers. The result is in ACC or P.
long **__mpyxu(** int *src1* **,** uint *src2* **);**	**MPYXU ACC, T,** {*mem*|*reg*}	The T register is loaded with *src1*. The *src2* is referenced by memory or loaded into a register. The result is in ACC.
long *dst* **= __norm32(**long *src* **,** int * *shift* **);**	**CSB ACC** **LSLL ACC, T** **MOV @** *shift* **, T**	Normalize *src* into *dst* and update **shift* with the number of bits shifted.
long long *dst* **= __norm64(**long long *src* **,** int * *shift* **);**	**CSB ACC** **LSL64 ACC:P, T** **MOV @** *shift* **, T** **CSB ACC** **LSL64 ACC:P, T** **MOV TMP16, AH** **MOV AH, T** **ADD** *shift* **, AH** **MOV AH, TMP16**	Normalize 64-bit *src* into *dst* and update **shift* with the number of bits shifted.
void **__or(** int * *m* **,** int *b* **);**	**OR** * *m* **,** *b*	OR the contents of memory location *m* to *b* and store the result in m, in an atomic way.
long **__qmpy32(** long *src32a* **,** long *src32b* **,** int *q* **);**	**CLRC OVM SPM − 1** **MOV T,** *src32a* **+ 1** **MPYXU P, T,** *src32b* **+ 0** **MOVP T,** *src32b* **+ 1** **MPYXU P, T,** *src32a* **+ 0** **MPYA P, T,** *src32a* **+ 1**	Extended precision DSP Q math. Different code is generated based on the value of *q*.
If *q* = 31,30:	**SPM** *q* **− 30** **SFR ACC, #45 −** *q* **ADDL ACC, P**	
If *q* = 29:	**SFR ACC, #16** **ADDL ACC, P**	
If *q* = 28 through 24:	**SPM** *q* **- 30** **SFR ACC, #16** **SFR ACC, #29 -** *q* **ADDL ACC, P**	
If *q* = 23 through 13:	**SFR ACC, #16** **ADDL ACC, P** **SFR ACC, #29 −** *q*	
If *q* = 12 through 0:	**SFR ACC, #16** **ADDL ACC, P** **SFR ACC, #16** **SFR ACC, #13 −** *q*	
long **__qmpy32by16(**long *src32* **,** int *src16* **,** int *q* **);**	**CLRC OVM** **MOV T,** *src16* **+ 0** **MPYXU P, T,** *src32* **+ 0** **MPY P, T,** *src32* **+ 1**	Extended precision DSP Q math. Different code is generated based on the value of *q*.
If *q* = 31, 30:	**SPM** *q* **− 30** **SFR ACC, #46 −** *q* **ADDL ACC, P**	
If *q* = 29 through 14:	**SPM 0** **SFR ACC, #16** **ADDL ACC, P** **SFR ACC, #30 −** *q*	
If *q* = 13 through 0:	**SPM 0** **SFR ACC, #16** **ADDL ACC, P** **SFR ACC, #16** **SFR ACC, #14 −** *q*	

Table 7-6. TMS320C28x C/C++ Compiler Intrinsics (continued)

Intrinsic	Assembly Instruction(s)	Description
void **__restore_interrupts(**unsigned int *val* **);**	**PUSH** *val* **POP ST1**	Restore interrupts and set the interrupt vector to value *val*.
long **__rol(** long *src* **);**	**ROL ACC**	Rotate ACC left.
long **__ror(** long *src* **);**	**ROR ACC**	Rotate ACC right.
void * *result* **= __rpt_mov_imm(**void * *dst* , int *src* , int *count* **);**	**MOV** *result* , *dst* **MOV ARx**, *dst* **RPT #** *count* **|| MOV *XARx++, #** *src*	Move the *dst* register to the *result* register. Move the *dst* register to a temp (ARx) register. Copy the immediate *src* to the temp register *count* + 1 times. The *src* must be a 16-bit immediate. The *count* can be an immediate from 0 to 255 or a variable.
int **__rpt_norm_inc(** long *src* , int *dst* , int *count* **);**	**MOV ARx**, *dst* **RPT #** *count* **|| NORM ACC, ARx++**	Repeat the normalize accumulator value *count* + 1 times. The *count* can be an immediate from 0 to 255 or a variable.
int **__rpt_norm_dec(**long *src* , int *dst* , int *count* **);**	**MOV ARx**, *dst* **RPT #** *count* **|| NORM ACC, ARx--**	Repeat the normalize accumulator value *count* + 1 times. The *count* can be an immediate from 0 to 255 or a variable.
long **__rpt_rol(**long *src* , int *count* **);**	**RPT #** *count* **|| ROL ACC**	Repeat the rotate accumulator left *count* + 1 times. The result is in ACC. The *count* can be an immediate from 0 to 255 or a variable.
long **__rpt_ror(**long *src* , int *count* **);**	**RPT #** *count* **|| ROR ACC**	Repeat the rotate accumulator right *count* + 1 times. The result is in ACC. The *count* can be an immediate from 0 to 255 or a variable.
long **__rpt_subcu(**long *dst* , int *src* , int *count* **);**	**RPT** *count* **|| SUBCU ACC**, *src*	The *src* operand is referenced from memory or loaded into a register and used as an operand to the SUBCU instruction. The result is in ACC. The *count* can be an immediate from 0 to 255 or a variable. The instruction repeats *count* + 1 times.
unsigned long **__rpt_subcul(**unsigned long *num*, unsigned long *den*, unsigned long *&remainder*, int *count* **);**	**RPT** *count* **|| SUBCUL ACC,** *den*	Performs repeated conditional long subtraction as typically used in unsigned modulus division. Returns the quotient.
long **__sat(** long *src* **);**	**SAT ACC**	Load ACC with 32-bit *src*. The result is in ACC.
long **__sat32(** long *src* , long *limit* **);**	**SETC OVM** **ADDL ACC,** {*mem*|**P**} **SUBL ACC,** {*mem*|**P**} **SUBL ACC,** {*mem*|**P**} **ADDL ACC,** {*mem*|**P**} **CLRC OVM**	Saturate a 32-bit value to a 32-bit mask. Load ACC with *src*. Limit value is either referenced from memory or loaded into the P register. The result is in ACC.
long **__sathigh16(**long *src* , int *limit* **);**	**SETC OVM** **ADDL ACC,** {*mem*|**P**}**<<16** **SUBL ACC,** {*mem*|**P**}**<<16** **SUBL ACC,** {*mem*|**P**}**<<16** **ADDL ACC,** {*mem*|**P**}**<<16** **CLRC OVM** **SFR ACC,** *rshift*	Saturate a 32-bit value to 16-bits high. Load ACC with *src*. The *limit* value is either referenced from memory or loaded into register. The result is in ACC. The result can be right shifted and stored into an int. For example: `ivar=__sathigh16(lvar, mask)>>6;`

Table 7-6. TMS320C28x C/C++ Compiler Intrinsics (continued)

Intrinsic	Assembly Instruction(s)	Description
long **__satlow16(** long *src* **);**	**SETC OVM** **MOV T, #0xFFFF** **CLR SXM** ; if necessary **ADD ACC, T <<15** **SUB ACC, T <<15** **SUB ACC, T <<15** **ADD ACC, T <<15** **CLRC OVM**	Saturate a 32-bit value to 16-bits low. Load ACC with *src*. Load T register with #0xFFFF. The result is in ACC.
long **__sbbu(** long *src1* **,** uint *src2* **);**	**SBBU ACC,** *src2*	Subtract *src2* + logical inverse of C from ACC (*src1*). The result is in ACC.
void **__sub(** int * *m* **,** int *b* **);**	**SUB** * *m* **,** *b*	Subtract *b* from the contents of memory location *m* and store the result in m, in an atomic way.
long **__subcu(** long *src1* **,** int *src2* **);**	**SUBCU ACC,** *src2*	Subtract *src2* shifted left 15 from ACC (*src1*). The result is in ACC.
unsigned long **__subcul(**unsigned long *num*, unsigned long *den*, unsigned long *&remainder* **);**	**SUBCUL ACC,** *den*	Performs a single conditional long subtraction as typically used in unsigned modulus division. Returns the quotient.
void **__subl(** long * *m* **,** long *b* **);**	**SUBL** * *m* **,** *b*	Subtract *b* from the contents of memory location *m* and store the result in m, in an atomic way.
void **__subr(** int * *m* **,** int *b* **);**	**SUBR** * *m* **,** *b*	Subtract the contents of memory location *m* from *b* and store the result in m, in an atomic way.
void **__subrl(** long * *m* **,** long *b* **);**	**SUBRL** * *m* **,** *b*	Subtract the contents of memory location *m* from *b* and store the result in m, in an atomic way.
if (__tbit(int *src* **,** int *bit* **));**	**TBIT** *src* **, #** *bit*	SET TC status bit if specified *bit* of *src* is 1.
float **__u32_bits_as_f32(** uint32_t *src*);	--	Packs a 32-bit register as a float. This intrinsic generates no code; it tells the compiler to change how it interprets the bits in place. See Section 7.6.1 for examples.
double **__u64_bits_as_f64(** uint64_t *src*);	--	Packs a 64-bit register as a double. This intrinsic generates no code; it tells the compiler to change how it interprets the bits in place. See Section 7.6.1 for examples.
void **__xor(** int * *m* **,** int *b* **);**	**XOR** * *m* **,** *b*	XOR the contents of memory location *m* to *b* and store the result in m, in an atomic way.

7.6.1 Floating Point Conversion Intrinsics

The following intrinsics extract the bits in a floating value or pack bits into a floating value. No instructions or movement of data is performed; these intrinsics tell the compiler how to interpret the bits in place. See Section 7.6 for calling syntax.

- __f32_bits_as_u32: Extract the bits in a float as a 32-bit register.
- __f64_bits_as_u64: Extracts the bits in a double as a 64-bit register.
- __u32_bits_as_f32: Packs a 32-bit register as a float.
- __u64_bits_as_f64: Packs a 64-bit register as a double.

The following function extracts the sign bit of a float. It does this by using the __f32_bits_as_u32 intrinsic to interpret a float as a 32-bit register. This intrinsic does not move the bits stored in *x*; it tells the compiler to change how it interprets the bits in place.

```
bool sign_bit_set(float x) {
    return __f32_bits_as_u32(x) >> 31;
}
```

Another way to use these intrinsics is for converting a float constant that has more precision than needed to a value that has all zero bits in the least significant mantissa bits. This enables the compiler to use one instruction instead of two to load a 32-bit constant into a register. The following code truncates a constant:

```
__u32_bits_as_f32(__f32_bits_as_u32(3.14159f) & 0xffff0000)
```

7.6.2 Floating Point Unit (FPU) Intrinsics

These intrinsics perform faster floating point calculations using the 32-bit (FPU32) and 64-bit (FPU64) hardware. These intrinsics are enabled if the --float_support compiler option is set to fpu32 or fpu64, respectively.

If you are using the COFF ABI, arguments and return values listed as floats may also use doubles, because both are 32-bit types. The FPU32 intrinsics are supported, but the FPU64 intrinsics are *not* supported if you are using the COFF ABI. An error occurs if you use the FPU64 intrinsics with long doubles when using COFF.

If you are using EABI, the float type is 32-bit and the double type is 64-bit. If FPU32 is enabled, use only the FPU32 versions. In general, if FPU64 is enabled, you may use both FPU32 and FPU64 intrinsics. However, in EABI mode with FPU64, there is no instruction that swaps two 32-bit floats. On FPU32 hardware, the SWAPF assembly instruction works only on 32-bit floats and the __swapf and __swapff intrinsics are equivalent. On FPU64 hardware, the SWAPF instruction works only 64-bit doubles; the __swapf intrinsic is supported but the __swapff intrinsic causes an error.

See Section 7.6.1 for intrinsics that reinterpret floats and doubles as registers (and vice versa).

Table 7-7. C/C++ Compiler Intrinsics for FPU

FPU Version	Intrinsic	Assembly Instruction(s)	Description
FPU32	float **__einvf32(** float *x* **);**	**EINVF32** *x*	Compute and return 1/*x* to about 8 bits of precision.
FPU64	double **__einvf64(** double *x* **);**	**EINVF64** *x*	Compute and return 1/*x* to about 8 bits of precision.
FPU32	float **__eisqrtf32(** float *x* **);**	**EISQRTF32** *x*	Find the square root of 1/*x* to about 8 bits of precision.
FPU64	double **__eisqrtf64(** double *x* **);**	**EISQRTF64** *x*	Find the square root of 1/*x* to about 8 bits of precision.
FPU32	void **__f32_max_idx(** float &*dst* , float *src* , float &*idx_dst* , float *idx_src* **);**	**MAXF32** *dst* , *src* **|| MOV32** *idx_dst* , *idx_src*	If *src*>*dst*, copy src to dst, and copy *idx_src* to *idx_dst*.
FPU64	void **__f64_max_idx(** double &*dst* , double *src* , double &*idx_dst* , double *idx_src* **);**	**MAXF64** *dst* , *src* **|| MOV64** *idx_dst* , *idx_src*	If *src*>*dst*, copy src to dst, and copy *idx_src* to *idx_dst*.
FPU32	void **__f32_min_idx(** float &*dst* , float *src* , float &*idx_dst* , float *idx_src* **);**	**MINF32** *dst* , *src* **|| MOV32** *idx_dst* , *idx_src*	If *src*<*dst*, copy src to dst, and copy *idx_src* to *idx_dst*.
FPU64	void **__f64_min_idx(** double &*dst* , double *src* , double &*idx_dst* , double *idx_src* **);**	**MINF64** *dst* , *src* **|| MOV64** *idx_dst* , *idx_src*	If *src*<*dst*, copy src to dst, and copy *idx_src* to *idx_dst*.
FPU32	int **__f32toi16r(**float *src* **);**	**F32TOI16R** *dst* , *src*	Convert float to int and round.
FPU32	unsigned int **__f32toui16r(**float *src* **);**	**F32TOUI16R** *dst* , *src*	Convert float to unsigned int and round.
FPU32	float **__fmax(** float *x* , float *y* **);**	**MAXF32** *y* , *x*	If *x*>*y*, copy *x* to *y*.
FPU64	double **__fmax64(** double *x* , double *y* **);**	**MAXF64** *y* , *x*	If *src*>*dst*, copy *x* to *y*.
FPU32	float **__fmin(** float *x* , float *y* **);**	**MINF32** *y* , *x*	If *x*<*y*, copy *x* to *y*.
FPU64	double **__fmin64(** double *x* , double *y* **);**	**MINF64** *y* , *x*	If *src*<*dst*, copy *x* to *y*.
FPU32	float **__fracf32(**float *src* **);**	**FRACF32** *dst* , *src*	Return the fractional portion of *src*.
FPU64	double **__fracf64(**double *src* **);**	**FRACF64** *dst* , *src*	Return the fractional portion of *src*.
FPU32	float **__fsat(**float *val* , float *max* , float *min* **);**	**MAXF32** *dst* , *min* **MINF32** *dst* , *max*	Return val if *min* < *val* < *max*. Else return *min* if *val* < *min*. Else return *max* if *val* > *max*. Calling __fsat with *max* < *min* results in undefined behavior.
FPU32 (only)	void **__swapff(** float &*a* , float &*b* **);**	**SWAPF** *a* , *b*	Swap the contents of *a* and *b*.
FPU32 (only)	void **__swapf(** float &*a* , float &*b* **);**	**SWAPF** *a* , *b*	Swap the contents of *a* and *b*.

Table 7-7. C/C++ Compiler Intrinsics for FPU (continued)

FPU Version	Intrinsic	Assembly Instruction(s)	Description
FPU64	void **__swapf(** double &*a* , double &*b* **);**	**SWAPF** *a* , *b*	Swap the contents of *a* and *b*.

7.6.3 Trigonometric Math Unit (TMU) Intrinsics

The following intrinsics perform faster trigonometric calculations using the Trigonometric Math Unit (TMU). These intrinsics are enabled if the --tmu_support=tmu0 compiler option is used. The shaded rows list intrinsics that are supported only if --tmu_support=tmu1, and are supported for EABI only.

If you are using the COFF ABI, arguments and return values listed as floats may also use doubles, because both are 32-bit types. If you are using EABI, these functions require the float type, because double is a 64-bit type.

Table 7-8. C/C++ Compiler Intrinsics for TMU

Intrinsic	Assembly Instruction(s)	Description
float **__atan(** float *src* **);**	**ATANPUF32** *dst* , *src* **MPY2PIF32** *dst* , *src*	Return the principal value of the arc tangent of *src* radians.
float **__atanpuf32(** float *src* **);**	**ATANPUF32** *dst* , *src*	Return the principal value of the arc tangent of *src*, which is provided as a per unit value.
float **__atan2(** float *y* , float *x* **);**	**QUADF32** *quadrant* , *ratio* , *y* , *x* **ATANPUF32** *atanpu* , *ratio* **ADDF32** *atan2pu* , *atanpu* **MPY2PIF32** *atan2* , *atan2pu*	Return the principal value of the arc tangent plus the quadrant for *x*, *y*.
float **__atan2puf32(** float *y*, float *x* **);**	**QUADF32** *quadrant* , *ratio* , *y* , *x* **ATANPUF32** *atanpu* , *ratio* **ADDF32** *dst* , *atanpu*	Return the principal value of the arc tangent plus the quadrant value for *y*, *x*. The value is returned as a per unit value.
float **__cos(** float *src* **);**	**DIV2PIF32** *dst* , *src* **COSPUF32** *dst* , *src*	Return the cosine of *src*, where *src* is provided in radians.
float **__cospuf32(** float *src* **);**	**COSPUF32** *dst* , *src*	Return the cosine of *src*, where *src* is provided as a per unit value.
float **__divf32(** float *num* , float *denom* **);**	**DIVF32** *dst* , *num* , *denom*	Return *num* divided by *denom* using the TMU hardware instruction for floating point division.
float **__div2pif32(** float *src* **);**	**DIV2PIF32** *dst* , *src*	Return the result of multiplying *src* by 1/2pi (effectively dividing by 2pi). This converts a value in radians to a per unit value.
float **__iexp2(** float *x* **);**	**IEXP2F32** *result* , *x*	Return the result of 2^^-|x|, which is the same as (1.0 / 2^^|x|). (tmu1 and EABI only)
float **__log2(** float *x* **);**	**LOG2F32** *logarithm* , *x*	Return the binary logarithm, which is the power to which the number 2 must be raised to obtain the value x. (tmu1 and EABI only)
float **__mpy2pif32(** float *src* **);**	**MPY2PIF32** *dst* , *src*	Return the result of multiplying *src* by 2pi. This converts a per unit value to radians. Per unit values are commonly used in control applications to represent normalized radians.
float **__quadf32(** float *ratio*, float *y*, float *x* **);**	**QUADF32** *quadrant* , *ratio* , *y* , *x*	Return the quadrant value (0.0, +/-0.25, or +/-0.5) and the ratio of *x* and *y*, which are provided as per unit values.
float **__sin(** float *src* **);**	**DIV2PIF32** *dst* , *src* **SINPUF32** *dst* , *src*	Return the sine of *src*, where *src* is provided in radians.
float **__sinpuf32(** float *src* **);**	**SINPUF32** *dst* , *src*	Return the sine of *src*, where *src* is provided as a per unit value.
float **__sqrt(** float *src* **);**	**SQRTF32** *dst* , *src*	Return the square root of *src*.

7.6.4 Fast Integer Division Intrinsics

The intrinsics listed in Table 7-9 perform faster division using hardware fast integer division support. These intrinsics are enabled if the --idiv_support=idiv0 compiler option is used.

In order to use these intrinsics, your code must include the `stdlib.h` header file and the --float_support option must be set to fpu32 or fpu64. Fast integer division support is available for EABI only.

These intrinsics follow the format of the ldiv and lldiv standard library functions. They take as input the *dividend* and *divisor*, and return a structure containing both the quotient and remainder in the fields *quot* and *rem*, respectively. The __uldiv_t and __ulldiv_t types are unsigned equivalents to ldiv and lldiv provided by `stdlib.h` . In addition, the following additional structure types are provided to return long long quotients with a long remainder:

```
typedef struct { long long quot; long rem; }                    __llldiv_t;
typedef struct { unsigned long long quot; unsigned long rem; } __ullldiv_t;
```

These intrinsics support three types of integer division:

- **Traditional division.** Also called truncated division or C99 division. This type of division rounds toward zero. As a consequence, the remainder sign is the same as the sign of the dividend.
- **Modulo division.** Also called floored division. This type of division rounds toward negative infinity. As a consequence, the remainder sign is the same as the sign of the divisor.
- **Euclidean division.** This type of division rounds toward infinite magnitude. As a consequence, the remainder is always non-negative.

For all three types of division, the *dividend* is exactly equal to *quotient* * *divisor* + *remainder*. All three types of division compute the same quotient and remainder if the remainder is zero or if the dividend and divisor are positive.

Since the results of division between two unsigned values do not vary for the traditional, Euclidean, and modulo versions, only the traditional versions are provided for unsigned inputs.

In general, if the dividend and divisor are of different types, the divisor is converted to the type of the dividend before division. However, in cases where the dividend type is signed, the divisor type is unsigned, and the dividend type is not larger than the divisor type—that is, for the *_div_i32byu32 and *_div_i64byu64 intrinsics—division is carried out using a signed type larger than both the dividend and divisor, and then converted to the type of the dividend.

Division by 0 is always undefined.

In addition to these intrinsics, when the --idiv_support=idiv0 compiler option is used, the built-in integer division and modulo operators ("/" and "%") use the appropriate faster instructions as described in Section 7.8.2. The faster versions of these built-in operators are used whether or not the `stdlib.h` header file is included.

Table 7-9. C/C++ Compiler Intrinsics for Fast Integer Division (--idiv_support=idiv0)

Intrinsic	Assembly Instruction(s)	Description
16-bit by 16-bit		
ldiv_t **__traditional_div_i16byi16(** int *dividend*, int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `I16TOF32 R1H,@Num` `MPYF32 R1H,R1H,#65536.0` `NOP` `F32TOI32 R1H,R1H` `ABSI32DIV32 R2H, R1H, R3H` `.loop #4 {SUBC4UI32 R2H, R1H, R3H}` `NEGI32DIV32 R1H, R2H`	Return the result of traditional 16-bit by 16-bit division.

Table 7-9. C/C++ Compiler Intrinsics for Fast Integer Division (--idiv_support=idiv0) (continued)

Intrinsic	Assembly Instruction(s)	Description
ldiv_t **__euclidean_div_i16byi16(** int *dividend*, int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `I16TOF32 R1H,@Num` `MPYF32 R1H,R1H,#65536.0` `NOP` `F32TOI32 R1H,R1H` `ABSI32DIV32 R2H, R1H, R3H` `.loop #4 {SUBC4UI32 R2H, R1H, R3H}` `ENEGI32DIV32 R1H, R2H, R3H`	Return the result of Eucildean 16-bit by 16-bit division.
ldiv_t **__modulo_div_i16byi16(** int *dividend*, int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `I16TOF32 R1H,@Num` `MPYF32 R1H,R1H,#65536.0` `NOP` `F32TOI32 R1H,R1H` `ABSI32DIV32 R2H, R1H, R3H` `.loop #4 {SUBC4UI32 R2H, R1H, R3H}` `MNEGI32DIV32 R1H, R2H, R3H`	Return the result of modulo 16-bit by 16-bit division.
__uldiv_t **__traditional_div_u16byu16(** unsigned int *dividend*, unsigned int *divisor* **);**	`UI16TOF32 R3H,@Den` `F32TOUI32 R3H,R3H` `UI16TOF32 R1H,@Num` `MPYF32 R1H,R1H,#65536.0` `NOP` `F32TOUI32 R1H,R1H` `.loop #4{SUBC4UI32 R2H, R1H, R3H}`	Return the unsigned result of traditional 16-bit by 16-bit division, when the dividend and divisor are unsigned.
32-bit by 32-bit		
ldiv_t **__traditional_div_i32byi32(** long *dividend*, long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ABSI32DIV32 R2H, R1H, R3H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `NEGI32DIV32 R1H, R2H`	Return the result of traditional 32-bit by 32-bit division.
ldiv_t **__euclidean_div_i32byi32(** long *dividend*, long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ABSI32DIV32 R2H, R1H, R3H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `ENEGI32DIV32 R1H, R2H, R3H`	Return the result of Eucildean 32-bit by 32-bit division.
ldiv_t **__modulo_div_i32byi32(** long *dividend*, long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ABSI32DIV32 R2H, R1H, R3H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `MNEGI32DIV32 R1H, R2H, R3H`	Return the result of modulo 32-bit by 32-bit division.
ldiv_t **__traditional_div_i32byu32(** long *dividend*, unsigned long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ABSI32DIV32U R2H, R1H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `NEGI32DIV32 R1H, R2H`	Return the result of traditional 32-bit by 32-bit division, when the divisor is unsigned.
ldiv_t **__euclidean_div_i32byu32(** long *dividend*, unsigned long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ABSI32DIV32U R2H, R1H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `ENEGI32DIV32 R1H, R2H, R3H`	Return the result of Eucildean 32-bit by 32-bit division, when the divisor is unsigned.

Table 7-9. C/C++ Compiler Intrinsics for Fast Integer Division (--idiv_support=idiv0) (continued)

Intrinsic	Assembly Instruction(s)	Description
ldiv_t **__modulo_div_i32byu32(** long *dividend*, unsigned long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ABSI32DIV32U R2H, R1H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `MNEGI32DIV32 R1H, R2H, R3H`	Return the result of modulo 32-bit by 32-bit division, when the divisor is unsigned.
__uldiv_t **__traditional_div_u32byu32(** unsigned long *dividend*, unsigned long *divisor* **);**	`MOV32 R3H @DEN` `MOV32 R1H @NUM` `ZERO R2` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}`	Return the unsigned result of traditional 32-bit by 32-bit division, when the dividend and divisor are unsigned.
32-bit by 16-bit		
ldiv_t **__traditional_div_i32byi16(** long *dividend*, int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `MOV R1H, @NUM` `.loop #3 {NOP}` `ABSI32DIV32 R2H, R1H, R3H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `NEGI32DIV32 R1H, R2H`	Return the result of traditional 32-bit by 16-bit division.
ldiv_t **__euclidean_div_i32byi16(** long *dividend*, int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `MOV R1H, @NUM` `.loop #3 {NOP}` `ABSI32DIV32 R2H, R1H, R3H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `ENEGI32DIV32 R1H, R2H, R3H`	Return the result of Eucildean 32-bit by 16-bit division.
ldiv_t **__modulo_div_i32byi16(** long *dividend*, int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `MOV R1H, @NUM` `.loop #3 {NOP}` `ABSI32DIV32 R2H, R1H, R3H` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}` `MNEGI32DIV32 R1H, R2H, R3H`	Return the result of modulo 32-bit by 16-bit division.
__uldiv_t **__traditional_div_u32byu16(** unsigned long *dividend*, unsigned int *divisor* **);**	`I16TOF32 R3H,@Den` `F32TOI32 R3H,R3H` `MOV R1H, @NUM` `ZERO R2` `.loop #2 {NOP}` `.loop #8 {SUBC4UI32 R2H, R1H, R3H}`	Return the unsigned result of traditional 32-bit by 16-bit division, when the dividend and divisor are unsigned.
64-bit by 64-bit		
lldiv_t **__traditional_div_i64byi64(** long long *dividend*, long long *divisor* **);**	`MOV32 R5H @DEN_L` `MOV32 R3H @DEN_H` `MOV32 R0H @NUM_L` `MOV32 R1H @NUM_H` `ABSI64DIV64 R2H:R4H, R1H:R0H, R3H:R5H` `.loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H,` `R3H:R5H}` `NEGI64DIV64 R1H:R0H, R2H:R4H`	Return the result of traditional 64-bit by 64-bit division.
lldiv_t **__euclidean_div_i64byi64(** long long *dividend*, long long *divisor* **);**	`MOV32 R5H @DEN_L` `MOV32 R3H @DEN_H` `MOV32 R0H @NUM_L` `MOV32 R1H @NUM_H` `ABSI64DIV64 R2H:R4H, R1H:R0H, R3H:R5H` `.loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H,` `R3H:R5H}` `ENEGI64DIV64 R1H:R0H, R2H:R4H, R3H:R5H`	Return the result of Eucildean 64-bit by 64-bit division.

Table 7-9. C/C++ Compiler Intrinsics for Fast Integer Division (--idiv_support=idiv0) (continued)

Intrinsic	Assembly Instruction(s)	Description
lldiv_t **__modulo_div_i64byi64(** long long *dividend*, long long *divisor* **);**	MOV32 R5H @DEN_L MOV32 R3H @DEN_H MOV32 R0H @NUM_L MOV32 R1H @NUM_H ABSI64DIV64 R2H:R4H, R1H:R0H, R3H:R5H .loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H, R3H:R5H} MNEGI64DIV64 R1H:R0H, R2H:R4H, R3H:R5H	Return the result of modulo 64-bit by 64-bit division.
lldiv_t **__traditional_div_i64byu64(** long long *dividend*, unsigned long long *divisor* **);**	MOV32 R5H @DEN_L MOV32 R3H @DEN_H MOV32 R0H @NUM_L MOV32 R1H @NUM_H ABSI64DIV64U R2H:R4H, R1H:R0H .loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H, R3H:R5H} NEGI64DIV64 R1H:R0H, R2H:R4H	Return the result of traditional 64-bit by 64-bit division, when the divisor is unsigned.
lldiv_t **__euclidean_div_i64byu64(** long long *dividend*, unsigned long long *divisor* **);**	MOV32 R5H @DEN_L MOV32 R3H @DEN_H MOV32 R0H @NUM_L MOV32 R1H @NUM_H ABSI64DIV64U R2H:R4H, R1H:R0H .loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H, R3H:R5H} ENEGI64DIV64 R1H:R0H, R2H:R4H, R3H:R5H	Return the result of Eucildean 64-bit by 64-bit division, when the divisor is unsigned.
lldiv_t **__modulo_div_i64byu64(** long long *dividend*, unsigned long long *divisor* **);**	MOV32 R5H @DEN_L MOV32 R3H @DEN_H MOV32 R0H @NUM_L MOV32 R1H @NUM_H ABSI64DIV64U R2H:R4H, R1H:R0H .loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H, R3H:R5H} MNEGI64DIV64 R1H:R0H, R2H:R4H, R3H:R5H	Return the result of modulo 64-bit by 64-bit division, when the divisor is unsigned.
__ulldiv_t **__traditional_div_u64byu64(** unsigned long long *dividend*, unsigned long long *divisor* **);**	ZERO R2 ZERO R4 MOV32 R5H @DEN_L MOV32 R3H @DEN_H MOV32 R0H @NUM_L MOV32 R1H @NUM_H .loop #32 {SUBC2UI64 R2H:R4H, R1H:R0H, R3H:R5H}	Return the unsigned result of traditional 64-bit by 64-bit division, when the dividend and divisor are unsigned.
64-bit by 32-bit		
__lldiv_t **__traditional_div_i64byi32(** signed long long *dividend*, long *divisor* **);**	ABSI64DIV32 R2H, R1H:R0H, R3H .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 NEGI64DIV32 R1H:R0H, R2H LRETR	Return the result of traditional 64-bit by 32-bit division.
__lldiv_t **__euclidean_div_i64byi32(** signed long long *dividend*, long *divisor* **);**	ABSI64DIV32 R2H, R1H:R0H, R3H .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 ENEGI64DIV32 R1H:R0H, R2H, R3H LRETR	Return the result of Eucildean 64-bit by 32-bit division.

Table 7-9. C/C++ Compiler Intrinsics for Fast Integer Division (--idiv_support=idiv0) (continued)

Intrinsic	Assembly Instruction(s)	Description
__lldiv_t **__modulo_div_i64byi32(** signed long long *dividend*, long *divisor* **);**	ABSI64DIV32 R2H, R1H:R0H, R3H .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 MNEGI64DIV32 R1H:R0H, R2H, R3H LRETR	Return the result of modulo 64-bit by 32-bit division.
__lldiv_t **__traditional_div_i64byu32(** signed long long *dividend*, unsigned long *divisor* **);**	ABSI64DIV32U R2H, R1H:R0H .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 ZERO R4 SWAPF R4, R2 NEGI64DIV64 R1H:R0H, R2H:R4H LRETR	Return the result of traditional 64-bit by 32-bit division, when the divisor is unsigned.
__lldiv_t **__euclidean_div_i64byu32(** signed long long *dividend*, unsigned long *divisor* **);**	ABSI64DIV32U R2H, R1H:R0H .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 ENEGI64DIV32 R1H:R0H, R2H, R3H LRETR	Return the result of Eucildean 64-bit by 32-bit division, when the divisor is unsigned.
__lldiv_t **__modulo_div_i64byu32(** unsigned long long *dividend*, unsigned long *divisor* **);**	ABSI64DIV32U R2H, R1H:R0H .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 MNEGI64DIV32 R1H:R0H, R2H, R3H LRETR	Return the result of modulo 64-bit by 32-bit division, when the divisor is unsigned.
__ulldiv_t **__traditional_div_u64byu32(** unsigned long long *dividend*, unsigned long *divisor* **);**	ZERO R2 ZERO R4 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 .loop 8 SUBC4UI32 R2H, R1H, R3H .endloop SWAPF R1, R0 LRETR	Return the unsigned result of traditional 64-bit by 32-bit division, when the dividend and divisor are unsigned.

7.7 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.

7.7.1 General Points About Interrupts

An interrupt routine can perform any task performed by any other function, including accessing global variables, allocating local variables, and calling other functions.

When you write interrupt routines, keep the following points in mind:

- An interrupt handling routine cannot have arguments. If any are declared, they are ignored.
- An interrupt handling routine can be called by normal C/C++ code, but it is inefficient to do this because all the registers are saved.
- An interrupt handling routine can handle a single interrupt or multiple interrupts. The compiler does not generate code that is specific to a certain interrupt, except for c_int00, which is the system reset interrupt. When you enter this routine, you cannot assume that the run-time stack is set up; therefore, *you cannot allocate local variables, and you cannot save any information on the run-time stack*.
- To associate an interrupt routine with an interrupt, the address of the interrupt function must be placed in the appropriate interrupt vector. You can use the assembler and linker to do this by creating a simple table of interrupt addresses using the .sect assembler directive.
- In assembly language, remember to precede the symbol name with an underscore. For example, refer to c_int00 as _c_int00.

7.7.2 Using C/C++ Interrupt Routines

If a C/C++ interrupt routine does not call any other functions, only those registers that the interrupt handler uses 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 compiler saves all the save-on-call registers if any other functions are called.

However, if a --float_support=fpu32/fpu64 option is used, the STF register is saved and restored. Any STF flag modifications performed by the interrupt routine will be undone when the routine returns.

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. Interrupt handling functions should not be called directly.

Interrupts can be handled directly with C/C++ functions by using the INTERRUPT pragma or the __interrupt keyword. For information about the INTERRUPT pragma, see Section 6.9.15. For information about the __interrupt keyword, see Section 6.5.3.

7.8 Integer Expression Analysis

This section describes some special considerations to keep in mind when evaluating integer expressions.

7.8.1 Operations Evaluated With Run-Time-Support Calls

The TMS320C28x does not directly support some C/C++ integer operations. Evaluating these operations is done with calls to run-time-support routines. These routines are hard-coded in assembly language. They are members of the object and source run-time-support libraries (such as rts2800_ml.lib) in the toolset.

The conventions for calling these routines are modeled on the standard C/C++ calling conventions.

Operation Type	Operations Evaluated With Run-Time-Support Calls
16-bit int	Divide (signed)
	Modulus
32-bit long	Divide (signed)
	Modulus
64-bit long long	Multiply(1)
	Divide
	Bitwise AND, OR, and XOR
	Compare

(1) 64-bit long long multiplies are inlined if -mf=5 is specified.

7.8.2 Division Operations with Fast Integer Division Support

If the --idiv_support=idiv0 command-line option is used, the compiler generates faster instructions to perform integer division when the division ("/") or modulo ("%") operator or the div() or ldiv() function is used. The faster versions of these built-in operators are used whether or not the `stdlib.h` header file is included.

The following table shows which intrinsics are used to perform integer operations using the division ("/") and modulo ("%") operators.

Types Operated Upon	Equivalent Intrinsic Call
int / int	__traditional_div_i16byi16(int, int).quot
int % int	__traditional_div_i16byi16(int, int).rem
unsigned int / unsigned int	__traditional_div_u16byu16(unsigned int, unsigned int).quot
unsigned int % unsigned int	__traditional_div_u16byu16(unsigned int, unsigned int).rem
long / long	__traditional_div_i32byi32(long, long).quot
long % long	__traditional_div_i32byi32(long, long).rem
unsigned long / unsigned long	__traditional_div_u32byu32(unsigned long, unsigned long).quot
unsigned long % unsigned long	__traditional_div_u32byu32(unsigned long, unsigned long).rem
long long / long long	__traditional_div_i64byi64(long long, long long).quot
long long % long long	__traditional_div_i64byi64(long long, long long).rem
unsigned long long / unsigned long long	__traditional_div_u64byu64(unsigned long long, unsigned long long).quot
unsigned long long % unsigned long long	__traditional_div_u64byu64(unsigned long long, unsigned long long).rem

Intrinsics that allow you to more exactly specify the operation to be performed are listed in Section 7.6.4.

In C, when the operands of integer division or modulo operations have different types, the compiler automatically performs "integral promotion" (also called implicit type conversion). That is, the compiler inserts an implicit cast to convert to a common type, and then performs the operation in that type.

The intrinsic that is performed may be different than the one expected based on the types being divided. For example:

```
long           dividend_i32, quotient_i32;
unsigned long  divisor_u32,  quotient_u32;
int            divisor_i16;
/* uses __traditional_div_u32byu32, not __traditional_div_i32byu32 */
quotient_u32 = dividend_i32 / divisor_u32;
/* uses __traditional_div_i32byi32, not __traditional_div_i32byi16 */
quotient_i32 = dividend_i32 / divisor_i16;
```

Because integral promotion can be confusing, it is often best to avoid the issue by making sure the operand types agree, possibly by using a cast operator. For more about integral promotion, see *How to Write Multiplies Correctly in C Code* (SPRA683).

7.8.3 C/C++ Code Access to the Upper 16 Bits of 16-Bit Multiply

The following methods provide access to the upper 16 bits of a 16-bit multiply in C/C++ language:

- Signed-results method:

```
    int m1, m2;
    int result;
    result = ((long) m1 * (long) m2) >> 16;
```

- Unsigned-results method:

```
    unsigned m1, m2;
    unsigned result;
    result = ((unsigned long) m1 * (unsigned long) m2) >> 16;
```

Note

Danger of Complicated Expressions

The compiler must recognize the structure of the expression for it to return the expected results. Avoid complicated expressions such as the following example:

```
((long)((unsigned)((a*b)+c)<5)*(long)(z*sin(w)>6))>>16
```

7.9 Floating-Point Expression Analysis

With the COFF ABI, both float and double are represented using IEEE single-precision numbers (32-bit). Long double values are represented using IEEE double-precision numbers (64-bit).

With EABI, the float type is represented using IEEE single-precision numbers (32-bit). Both the double type and the long double type are represented using IEEE double-precision numbers (64-bit).

The run-time-support library, rts2800_ml.lib, contains a set of floating-point math functions that support:

- Addition, subtraction, multiplication, and division
- Comparisons (>, <, >=, <=, ==, !=)
- Conversions from integer or long to floating-point and floating-point to integer or long, both signed and unsigned
- Standard error handling

The conventions for calling these routines are the same as the conventions used to call the integer operation routines. Conversions are unary operations.

7.10 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 called 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.

The c_int00 function performs the following tasks to initialize the environment:

1. Defines a section called .stack for the system stack and sets up the initial stack pointers
2. Initializes global variables by copying the data from the initialization tables to the storage allocated for the variables in the .ebss or .bss section. If you are initializing variables at load time (--ram_model option), a loader performs this step before the program runs (it is not performed by the boot routine). For more information, see Section 7.10.3.
3. Executes the global constructors found in the global constructors table. For more information, see Section 7.10.3.4.
4. 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.

7.10.1 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.

7.10.2 Run-Time Stack

The run-time stack is allocated in a single continuous block of memory and grows down from low addresses to higher addresses. The SP points to the top of the stack.

The code does not check to see if the run-time stack overflows. Stack overflow occurs when the stack grows beyond the limits of the memory space that was allocated for it. Be sure to allocate adequate memory for the stack.

The stack size can be changed at link time by using the --stack_size link option on the linker command line and specifying the stack size as a constant directly after the option.

7.10.3 Automatic Initialization of Variables for COFF

Note

This section applies to applications that use the COFF ABI only.

Some global variables must have initial values assigned to them before a C/C++ program starts running. The process of retrieving these variables' data and initializing the variables with the data is called autoinitialization.

The compiler builds tables in a special section called .cinit that contains data for initializing global and static variables. Each compiled module contains these initialization tables. The linker combines them into a single table (a single .cinit section). The boot routine or a loader uses this table to initialize all the system variables.

Note

Initializing Variables

In ANSI/ISO C, global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ compiler does not perform any preinitialization of uninitialized variables. Explicitly initialize any variable that must have an initial value of 0.

The easiest method is to have a loader clear the .ebss or .bss section before the program starts running. Another method is to set a fill value of 0 in the linker control map for the .ebss or .bss section.

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

Global variables are either autoinitialized at run time or at load time; see Section 7.10.3.2 and Section 7.10.3.3. Also see Section 6.13.

7.10.3.1 Initialization Tables

The tables in the .cinit section consist of variable-size initialization records. Each variable that must be autoinitialized has a record in the .cinit section. Figure 7-2 shows the format of the .cinit section and the initialization records.

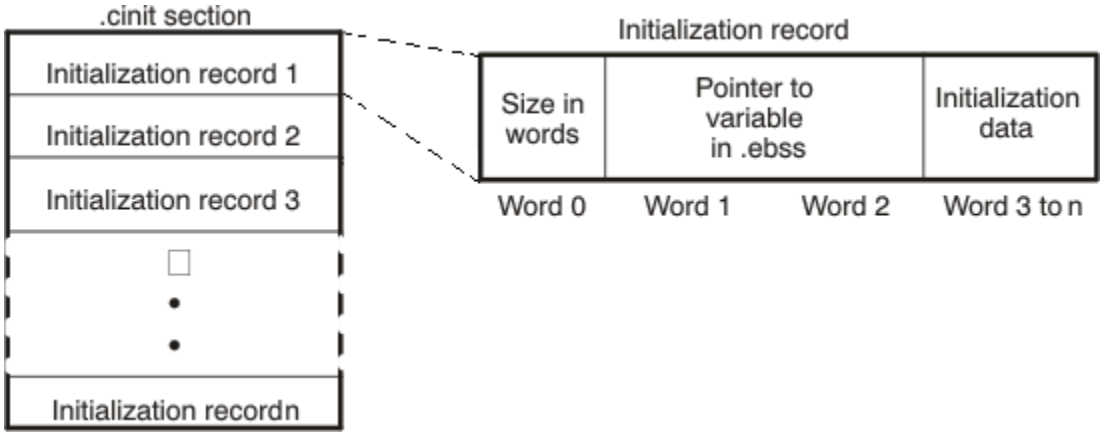


Figure 7-2. Format of Initialization Records in the .cinit Section

The fields of an initialization record contain the following information:

- The first field of an initialization record contains the size (in words) of the initialization data. That is, this field contains the size of the third field. The size is specified as a negative value; this is legacy behavior, and the absolute value of this field is the size of the data.
- The second field contains the starting address of the area within the .ebss (or .bss) section where the initialization data must be copied. The second field requires two words to hold the address.
- The third field contains the data that is copied into the .ebss (or .bss) section to initialize the variable. The width of this field is variable.

Each variable that must be autoinitialized has an initialization record in the .cinit section.

The following example shows initialized global variables defined in C.

```
int    i= 23;
int j[2] = { 1,2};
```

The corresponding initialization table is as follows:

```
        .global   _i
        .ebss     _i,1,1,0
        .global   _j
_j:     .usect    .ebss,2,1,0
        .sect     ".cinit"
        .align    1
        .field        1,16
        .field        _i+0,16
        .field        23,16         ; _i @ 0
        .sect     ".cinit"
        .align    1
        .field        -IR_1,16
        .field        _j+0,32
        .field        1,16          ; _j[0] @ 0
        .field        2,16          ; _j[1] @ 16
IR_1:   .set2
```

The .cinit section must contain only initialization tables in this format. When interfacing assembly language modules, do not use the .cinit section for any other purpose.

The table in the .pinit or .init_array section (depending on the ABI) simply consists of a list of addresses of constructors to be called (see Figure 7-3). The constructors appear in the table after the .cinit initialization.

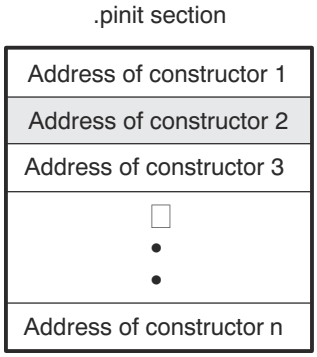


Figure 7-3. Format of Initialization Records in the .pinit or .init_array Section

When you use the --rom_model or --ram_model option, the linker combines the .cinit sections from all the C modules and appends a null word to the end of the composite .cinit section. This terminating record appears as a record with a size field of 0 and marks the end of the initialization tables.

Likewise, the --rom_model or --ram_model link option causes the linker to combine all of the .pinit or .init_array sections from all C/C++ modules and append a null word to the end of the composite .pinit or .init_array section. The boot routine knows the end of the global constructor table when it encounters a null constructor address.

The const-qualified variables are initialized differently; see Section 6.5.1.

7.10.3.2 Autoinitialization of Variables at Run Time for COFF

Note

This section applies to applications that use the COFF ABI only.

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the --rom_model option.

Using this method, the .cinit section is loaded into memory along with all the other initialized sections, and global variables are initialized at run time. The linker defines a special symbol called cinit that points to the beginning of the initialization tables in memory. When the program begins running, the C/C++ boot routine copies data from the tables (pointed to by .cinit) into the specified variables in the .ebss section. This allows initialization data to be stored in ROM and copied to RAM each time the program starts.

Figure 7-4 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

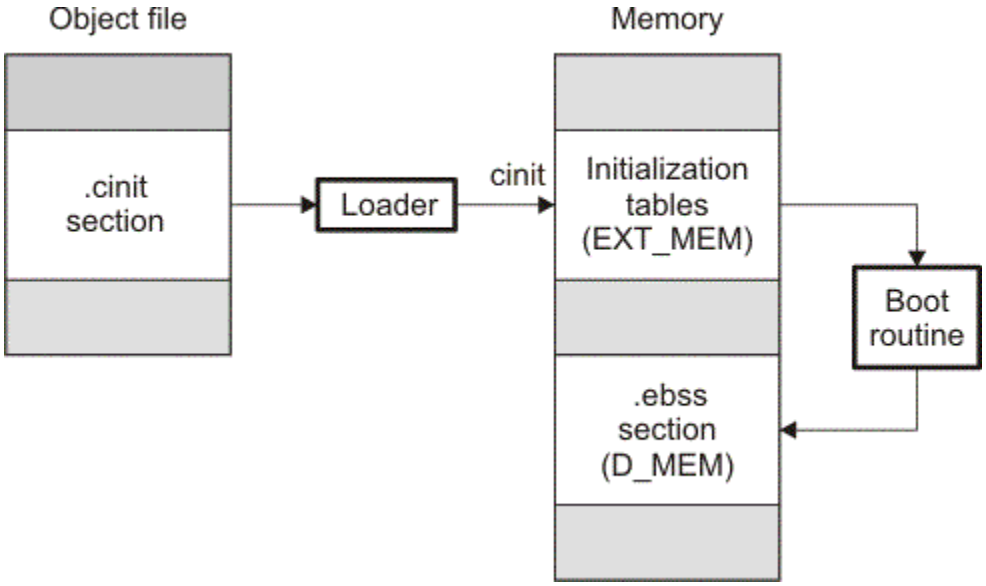


Figure 7-4. Autoinitialization at Run Time

7.10.3.3 Initialization of Variables at Load Time for COFF

Note

This section applies to applications that use the COFF ABI only.

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the --ram_model option.

When you use the --ram_model link option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to -1 (normally, cinit points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run-time initialization is performed at boot time.

A loader (which is not part of the compiler package) must be able to perform the following tasks to use initialization at load time:

- Detect the presence of the .cinit section in the object file
- Determine that STYP_COPY is set in the .cinit section header, so that it knows not to copy the .cinit section into memory
- Understand the format of the initialization tables

Figure 7-5 illustrates the initialization of variables at load time.

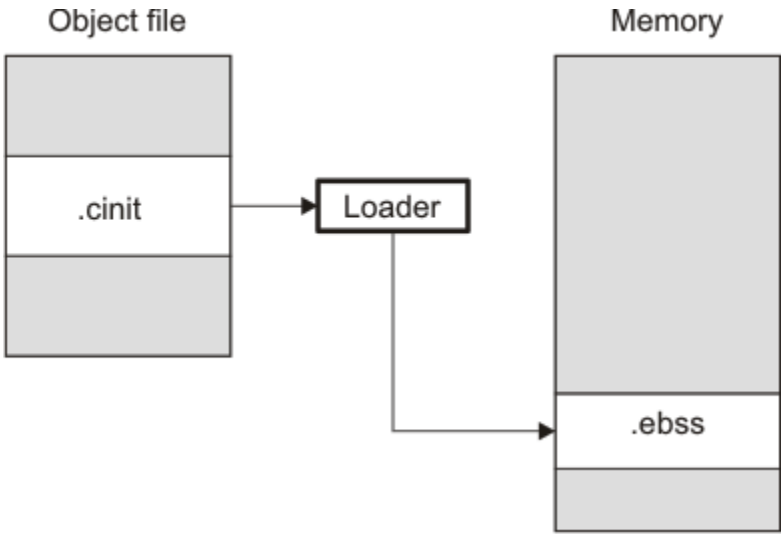


Figure 7-5. Initialization at Load Time

Regardless of the use of the --rom_model or --ram_model options, the .pinit or .init_array section is always loaded and processed at run time.

7.10.3.4 Global Constructors

Note

This section applies to applications that use the COFF ABI only.

All global C++ variables that have constructors must have their constructor called before main(). The compiler builds a table in a section called .pinit of global constructor addresses that must be called, in order, before main(). The linker combines the .pinit section form each input file to form a single table in the .pinit section. The boot routine uses this table to execute the constructors.

7.10.4 Automatic Initialization of Variables for EABI

Note

This section applies to applications that use EABI only.

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.

7.10.4.1 Zero Initializing Variables

Note

This section applies to applications that use EABI only.

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.

7.10.4.2 Direct Initialization for EABI

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 7.10.4.5.

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 7.10.4.3.

7.10.4.3 Autoinitialization of Variables at Run Time for EABI

Note

This section applies to applications that use EABI only.

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 7-6 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

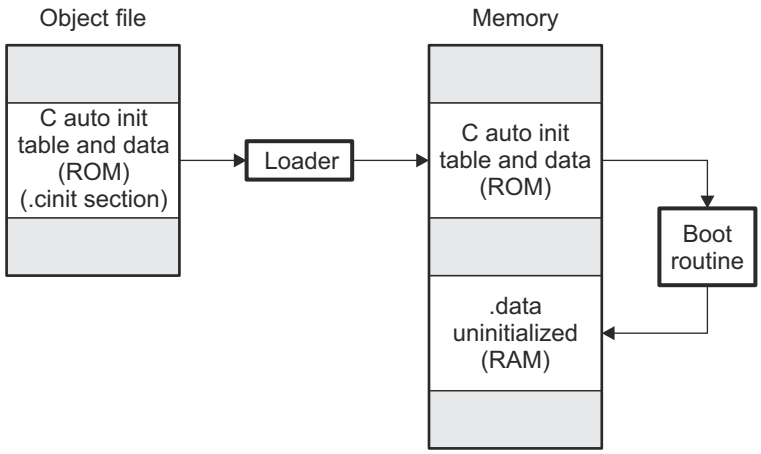


Figure 7-6. Autoinitialization at Run Time

7.10.4.4 Autoinitialization Tables for EABI

Note

This section applies to applications that use EABI only.

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.

Note

Migration from COFF to ELF Initialization

The name .cinit is used primarily to simplify migration from COFF to ELF format and the .cinit section created by the linker has nothing in common (except the name) with the COFF cinit records.

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:

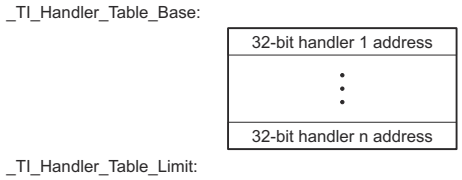
The linker defined symbols __TI_CINIT_Base and __TI_CINIT_Limit point to the start and end of the table, respectively. Each entry in this table corresponds to one output section that needs to be initialized. The initialization data for each output section could be encoded using different encoding.

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.



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.

7.10.4.4.1 Length Followed by Data Format

Note

This section applies to applications that use EABI only.

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.

7.10.4.4.2 Zero Initialization Format

Note

This section applies to applications that use EABI only.

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.

7.10.4.4.3 Run Length Encoded (RLE) Format

Note

This section applies to applications that use EABI only.

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.

7.10.4.4.4 Lempel-Ziv-Storer-Szymanski Compression (LZSS) Format

Note

This section applies to applications that use EABI only.

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.

The decompression algorithm for LZSS is as follows:

1. Read 16 bits, which are the encoding flags (F) marking the start of the next LZSS encoded packet.
2. For each bit (B) in F, starting from the least significant to the most significant bit, do the following:
   a. If (B & 0x1), read the next 16 bits and write it to the output buffer. Then advance to the next bit (B) in F and repeat this step.
   b. Else read the next 16-bits into temp (T), length (L) = (T & 0xf) + 2, and offset (O) = (T >> 4).
      i. If L == 17, read the next 16-bits (L'); then L += L'.
      ii. If O == LZSS end of data (LZSS_EOD), we've reached the end of the data, and the algorithm is finished.
      iii. At position (P) = output buffer - Offset (O) - 1, read L bytes from position P and write them to the output buffer.
      iv. Go to step 2a.

7.10.4.5 Initialization of Variables at Load Time

Note

This section applies to applications that use EABI only.

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 7-7 illustrates the initialization of variables at load time.

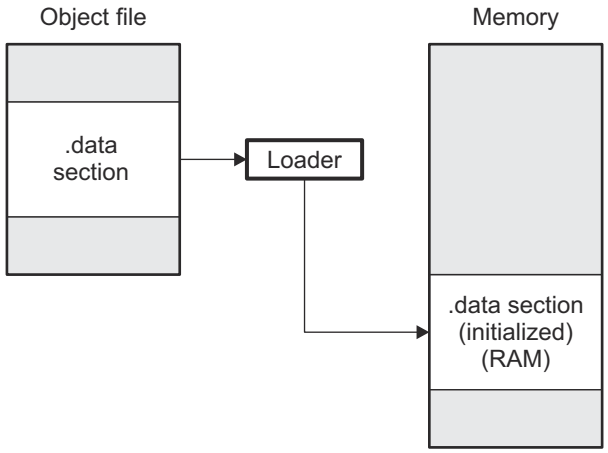


Figure 7-7. Initialization at Load Time

7.10.4.6 Global Constructors

Note

This section applies to applications that use EABI only.

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.

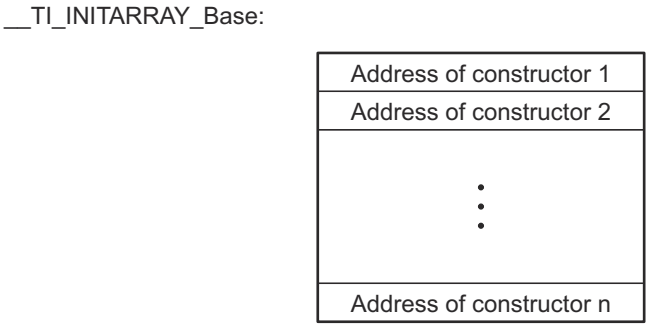


Figure 7-8. Constructor Table

This page intentionally left blank.

Chapter 8

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 exception conditions, 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 8.1 and Section 8.2.

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 8.5 .

8.1 C and C++ Run-Time Support Libraries

TMS320C28x compiler releases include pre-built run-time support (RTS) libraries that provide all the standard capabilities. Separate libraries are provided for FPU support and C++ exception support. See Section 8.1.8 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 6.1.

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

8.1.1 Linking Code With the Object Library

When you link your program, you must specify the object library as one of the linker input files so that references to the I/O and run-time-support functions can be resolved. You can either specify the library or allow the compiler to select one for you. See Section 4.3.1 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*TMS320C28x Assembly Language Tools User's Guide*.

C, C++, and mixed C and C++ programs can use the same run-time-support library. Run-time-support functions and variables that can be called and referenced from both C and C++ will have the same linkage.

8.1.2 Header Files

You must use the header files provided with the compiler run-time support when using functions from C/C++ standard library. Set the C2000_C_DIR environment variable to the specific include directory: "include\lib".

8.1.3 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\ c2000_#.#.# \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 rts2800_ml.lib library or to build a new library. See Section 8.1.8 for details on library naming and Section 8.5 for details on building

8.1.4 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

8.1.5 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 int. The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h> but does not include all the functions specified in the standard. See Section 6.4 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.

8.1.6 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 8-1. 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.

8.1.7 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.

8.1.8 Library Naming Conventions

By default, the linker uses automatic library selection to select the correct run-time-support library (see Section 4.3.1.1) for your application. If you select the library manually, you must select the matching library using a naming scheme like the following:

```
rts2800_[ml|fpu32|fpu64][_eabi][_eh].lib
```

The components of this naming convention are as follows:

rts2800	Indicates the library is built for C28x support.
_ml	Indicates the library contains no FPU support.
_fpu32	Indicates support for 32-bit FPU targets.
_fpu64	Indicates support for 64-bit FPU targets. (EABI only)
_eabi	Indicates the library is in EABI format. If the name does not contain "_eabi", the library is a COFF library.
_eh	Indicates the library provides exception handling support.

8.2 The C I/O Functions

The C I/O functions make it possible to access the host's operating system to perform I/O. The capability to perform I/O on the host gives you more options when debugging and testing code.

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 *TMS320C28x 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).

8.2.1 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 the file main.out from the run-time-support library:

```
cl2000 main.c --run_linker --heap_size=400 --library=rts2800_ml.lib --output_file=main.out
```

Executing main.out results in

```
Hello, world
```

being output to a file and

```
Hello again, world
```

being output to your host's stdout window.

8.2.1.1 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.

8.2.2 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 8.2.3 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 8.2.5).
- 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 8.2.3.

The *path* is the filename of the file, including path information and optional device prefix. (See Section 8.2.5.)

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.

8.2.3 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 8.2.5 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 8.2.3.

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 doesn't 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.

8.2.4 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 8-1. 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 8-1. 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 8-1. 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 */
    /* doesn't 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"); }
```

8.2.5 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 8.2.2. The device driver for the HOST that the TMS320C28x 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

Example

Example 8-2 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 8-2 illustrates adding and using a device for C I/O:

Example 8-2. 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);
}
```

8.3 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.

8.4 Reinitializing Variables During a Warm Start

The ability to update system firmware while the system is running and begin using the new firmware once the update is complete is called Live Firmware Update (LFU). This is also described as a "warm start". The actual warm start is performed by a custom entry point function.

To support the creation of such entry points, the compiler provides the __TI_auto_init_warm() RTS function. This function reinitializes all global and static variables that have the update attribute and are therefore contained in the .TI.update section. See Section 6.15.4. The syntax for calling this function is as follows:

```
void __TI_auto_init_warm();
```

The __TI_auto_init_warm() routine does not need to be called if no global or static symbols use the update attribute. The custom entry point function is responsible for setting up the stack pointer (SP) and then calling main(). For information and examples of such functions, see the *Live Firmware Update Reference Design with C2000 MCUs* (TIDUEY4) design guide.

See Section 2.15 for more about LFU, which is supported for EABI only.

8.5 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 rts2800_ml.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.

8.5.1 Required Non-Texas Instruments Software

To use the self-contained run-time-support build process to rebuild a library with custom options, the following 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.

8.5.2 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 8.5.2.2 for situations when you might want to do this.

8.5.2.1 Automatic Standard Library Rebuilding by the Linker

The linker looks for run-time-support libraries primarily through the C2000_C_DIR environment variable. Typically, one of the pathnames in C2000_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 C2000_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=rts2800_ml.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/rts2800_ml.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 *TMS320C28x 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 C2000_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 C2000_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 8.5.2.2 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.

8.5.2.2 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.)

8.5.2.2.1 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=rts2800_ml.lib
```

For C28x these are some of the libraries can be built. See Section 8.1.8 for the full RTS library naming options.

- rts2800_ml.lib (C/C++ run-time object library with COFF output)
- rts2800_fpu32.lib (C/C++ run-time object library for 32-bit FPU targets with COFF output)
- rts2800_fpu64_eabi.lib (C/C++ run-time object library for 64-bit FPU targets with EABI output)

8.5.2.2.2 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.

8.5.2.2.3 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 rts2800_ml.lib, change the working directory to the 'lib' directory and run the command:

```
mklib --pattern=rts2800_ml.lib --name=rts2800_debug.lib --install_to=$Project/Debug --
extra_options="-g"
```

8.5.2.2.4 The mklib Program Option Summary

Run the following command to see the full list of options. These are described in Table 8-2.

```
mklib --help
```

Table 8-2. 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=rts2800_ml.lib --index=$C_DIR/lib
```

To build a custom library that is just like rts2800_ml.lib, but has symbolic debugging support enabled:

```
mklib --pattern=rts2800_ml.lib --extra_options="-g" --index=$C_DIR/lib --install_to=$Project/Debug
--name=rts2800_debug.lib
```

8.5.3 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.

8.5.3.1 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.

8.5.3.2 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 C2000_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 8-2 without error, even if they do not do anything.

Chapter 9

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.

9.1 Invoking the C++ Name Demangler

The syntax for invoking the C++ name demangler is:

dem2000 [*options*] [*filenames*]

dem2000	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, dem2000 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:

--abi={eabi|coffabi}	Demangling of EABI identifiers is on by default.
--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)

9.2 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.

```
int compute(int val, int *err);
int foo(int val, int *err)
{
   static int last_err = 0;
   int result = 0
   if (last_err == 0)
   result = compute(val, &last_err);
   *err = last_err;
   return result;
}
```

The resulting assembly that is output by the compiler is as follows.

```
;***************************************************************
;* FNAME: _foo_FiPi                      FR SIZE:   4           *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  3 Auto,  0 SOE     *
;***************************************************************
_foo_FiPi:
        ADDB        SP,#4
        MOVZ        DP,#_last_err$1
        MOV         *-SP[1],AL
        MOV         AL,@_last_err$1
        MOV         *-SP[2],AR4
        MOV         *-SP[3],#0
        BF          L1,NEQ
        ; branch occurs
        MOVL        XAR4,#_last_err$1
        MOV         AL,*-SP[1]
        LCR         #_compute__FiPi
        ; call occurs [#_compute__FiPi]
        MOV         *-SP[3],AL
L1:
        MOVZ        AR6,*-SP[2]
        MOV         *+XAR6[0],*(0:_last_err$1)
        MOV         AL,*-SP[3]
        SUBB        SP,#4
        LRETR
        ; return occurs
```

Executing the C++ name demangler will demangle all names that it believes to be mangled. Enter:

```
dem2000 foo.asm
```

The result after running the C++ name demangler is as follows. The linknames in foo( ) and compute( ) are demangled.

```
;***************************************************************
;* FNAME: foo(int, int *)                FR SIZE: 4            *
;*                                                             *
;* FUNCTION ENVIRONMENT                                        *
;*                                                             *
;* FUNCTION PROPERTIES                                         *
;*                            0 Parameter,  3 Auto,  0 SOE     *
;***************************************************************
foo(int, int *):
        ADDB        SP,#4
        MOVZ        DP,#_last_err$1
        MOV         *-SP[1],AL
        MOV         AL,@_last_err$1
        MOV         *-SP[2],AR4
        MOV         *-SP[3],#0
        BF          L1,NEQ
        ; branch occurs
        MOVL        XAR4,#_last_err$1
        MOV         AL,*-SP[1]
        LCR         #compute(int, int *)
        ; call occurs [#compute(int, int *)]
        MOV         *-SP[3],AL
L1:
        MOVZ        AR6,*-SP[2]
        MOV         *+XAR6[0],*(0:_last_err$1)
        MOV         AL,*-SP[3]
        SUBB        SP,#4
        LRETR
        ; return occurs
```

This page intentionally left blank.

Chapter 10

CLA Compiler

The TMS320C28x Software Development toolset also includes support for compiling Control Law Accelerator (CLA) C code. Because of CLA architecture and programming environment constraints, the C language support has some restrictions that are detailed in Section 10.2.3.

As when including CLA assembly, compiled CLA code is linked together with C28x code to create a single executable.

10.1 How to Invoke the CLA Compiler

The Control Law Accelerator (CLA) compiler is also invoked using the cl2000 command. Files that have a .cla extension are recognized by the compiler as CLA C files. The shell invokes separate CLA versions of the compiler passes to generate CLA-specific code. The --cla_support option is also required to assemble the output from the CLA code generator.

If you use the --cla_default option, files with an extension of .c are also compiled as CLA files.

Support is provided for Type 0, Type 1, and Type 2 CLA.

The object files generated by the compile can then be linked with other C28x objects files to create a combined C28x/CLA program.

To invoke the CLA compiler, enter:

cl2000 --cla_support=[cla0|cla1|cla2] [*other options*] *file* **.cla**

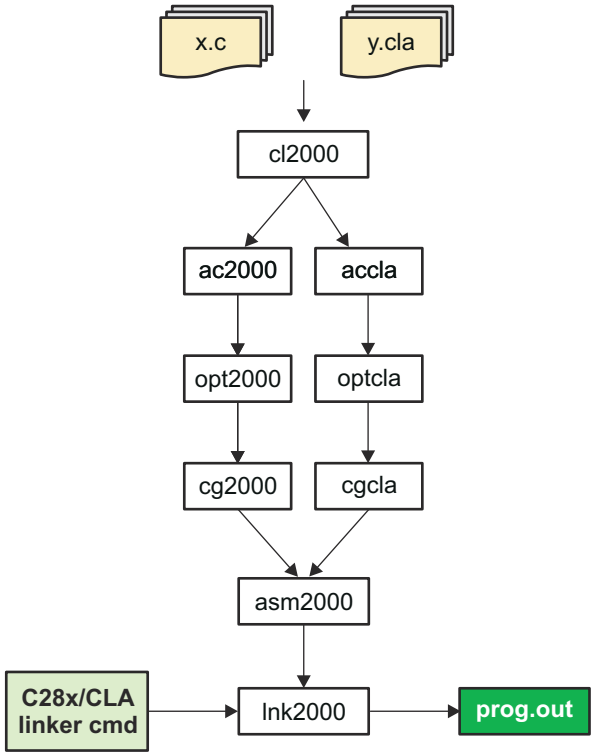


Figure 10-1. CLA Compilation Overview

Important facts about CLA C files:

- Files with .cla extension compile using a different parser, optimizer, and code generator.
- cl2000 does not support compiling files using both the C and CLA compilers in a single invocation.
- C28X/CLA builds requires linker command file changes to accommodate CLA compiler-generated sections and scratchpad area.

10.1.1 CLA-Specific Options

In addition to the --cla_support command-line option, the following command line options apply specifically to the CLA compiler:

--cla_default Causes files with an extension of .c to also be compiled as CLA files.

--cla_signed_compare_workaround={on|off} Enables the automatic use of a workaround for a CLA hardware flaw that affects integer comparisons. This is necessary because certain types of integer comparisons may produce incorrect results due to integer overflows, for example when the values compared have opposite signs and are near the extreme values. This option is off by default.

If you enable this option, a floating-point comparison is used internally to check the upper bits of the integer values being compared. This comparison detects whether an integer overflow may occur if the difference between the values being compared is too large. If an integer comparison may have an incorrect result, a floating-point comparison is performed instead. For a comparison like `if (x < y)` , the modified comparison performed is as follows:

```
(float)x < (float)y || (float)x == (float)y && (x <= y)
```

Note

Enabling this option increases code size and execution time if your code performs many 32-bit integer comparisons.

The following types of integer comparisons are always safe from integer overflows. The workaround is not used for such comparisons even if this option is enabled.

- Comparisons to zero
- Comparisons between two short ints.

Note than comparisons between unsigned integers are still subject to incorrect results, because the comparisons are performed internally in terms of signed integers.

If you do not want to use the workaround because of its effects on code size and execution time, you can use any of these manual coding alternatives:

- Identify known safe comparisons using the __mlt, __mleq, __mgt, __mgeq, __mltu, __mlequ, __mgtu, and __mgequ intrinsics, which are described in Table 10-2. These intrinsics indicate to the CLA compiler that the values being compared will not cause an overflow. For example, you can use the following code to perform such a comparison:

  ```
  if (__mlt(x, y))
  ```

- Cast the integers being compared as shorts if you know the values will always fit in a short. For example:

  ```
  if ((short)x < (short)y)
  for (short i = 0; i < (short)y); i++)
  ```

- Cast the integers being compared as floats if you know the values will not lose precision in a float. For example:

  ```
  if ((float)x < (float)y)
  for (float i = 0; i < (float)y); i++)
  ```

- Because comparisons to zero cannot overflow, you can rewrite loops as "down counters." For example:

  ```
  for (int i = y-1; i >= 0; i--)
  ```

10.2 CLA C Language Implementation

The CLA implementation supports C only. The compiler produces an error if compiling for C++.

The CLA C language implementation does not support the C standard library.

The CLA C language implementation requires changes from standard C. These are described in the subsections that follow.

10.2.1 Variables and Data Types

Note

All data shared between the CLA and C28x CPUs must be defined in the C28x C or C++ code, and not defined in the CLA code (that is, not in *.cla files). This is required because shared variables defined in CLA code may not be blocked properly, which may cause variable accesses on the C28x to not set the DP correctly. See Section 3.11 for information about blocking and the DP register for both COFF and EABI. The linker provides a diagnostic message if blocked data access is attempted on non-blocked data.

The following data types are supported:

Table 10-1. CLA Compiler Data Types

Type	EABI Size (bits)	COFF Size (bits)
char	16	16
short	16	16
int	32	32
long	32	32
long long	64	32
float	32	32
double	64	32
long double	64	32
pointer	16	16

A char/short should be limited to load/store operations.

Pointer sizes for CLA are always 16-bits. This differs from C28x, which has 32-bit pointers.

Note

CLA Has a Different Size for int

The size of an int for CLA is 32 bits instead of the 16-bit size on the C28x. To avoid ambiguity when sharing data between CLA and C28x, we strongly recommend that you use C99 type declarations that include size information (for example, int32_t and uint16_t).

Note

No 64-bit Types With CLA Compiler and COFF

When using the COFF ABI, the CLA compiler does not support 64-bit types.

10.2.2 Pragmas, Keywords, and Intrinsics

CLA accepts the C28x pragmas except for FAST_FUNC_CALL.

The far and ioport keywords are not recognized.

Access to the `'MMOV32 MSTF,mem32'` and `'MMOV32 mem32,MSTF'` instructions is provided using the cregister keyword. To access these MSTF instructions include the following declaration:

```
extern cregister volatile unsigned int MSTF;
```

The intrinsics listed in Table 10-2 are supported. Additionally, the run-time library functions abs() and fabs() are implemented as intrinsics.

Table 10-2. C/C++ Compiler Intrinsics for CLA

Intrinsic	Assembly Instruction(s)	Description
uint32_t **__f32_bits_as_u32(** float *src*);	--	Extracts the bits in a float as a 32-bit register. This intrinsic generates no code; it tells the compiler to change how it interprets the bits in place. See Section 7.6.1 for examples.
uint64_t **__f64_bits_as_u64(** double *src*);	--	Extracts the bits in a double as a 64-bit register. This intrinsic generates no code; it tells the compiler to change how it interprets the bits in place. See Section 7.6.1 for examples.
__mdebugstop()	**MDEBUGSTOP**	Debug stop
__meallow()	**MEALLOW**	Enable write access to EALLOW registers
__medis()	**MEDIS**	Disable write access to EALLOW registers
float **__meinvf32(**float *x* **);**	**MEINVF32** *x*	Estimate 1/*x* to about 8 bits of precision.
float **__meisqrtf32(**float *x* **);**	**MEISQRTF32** *x*	Estimate the square root of 1/*x* to about 8 bits of precision. The Newton-Raphson method is used to approximate the square root.
short **__mf32toi16r(**float *src* **);**	**MF32TOI16R** *dst* , *src*	Convert double to int and round.
unsigned short **__mf32toui16r(**float *src* **);**	**MF32TOUI16R** *dst* , *src*	Convert double to unsigned int and round.
float **__mfracf32(**float *src* **);**	**MFRACF32** *dst* , *src*	Return the fractional portion of *src*.
short **__mgeq(** signed int *x* , signed int *y* **);**	**MCMP32** *dst* , *src*	Return 1 if *x* is greater than or equal to *y*. Otherwise, return 0.
short **__mgequ(** unsigned int *x* , unsigned int *y* **);**	**MCMP32** *dst* , *src*	Return 1 if *x* is greater than or equal to *y*. Otherwise, return 0.
short **__mgt(** signed int *x* , signed int *y* **);**	**MCMP32** *dst* , *src*	Return 1 if *x* is greater than *y*. Otherwise, return 0.
short **__mgtu(** unsigned int *x* , unsigned int *y* **);**	**MCMP32** *dst* , *src*	Return 1 if *x* is greater than *y*. Otherwise, return 0.
short **__mleq(** signed int *x* , signed int *y* **);**	**MCMP32** *dst* , *src*	Return 1 if *x* is less than or equal to *y*. Otherwise, return 0.
short **__mlequ(** unsigned int *x* , unsigned int *y* **);**	**MCMP32** *dst* , *src*	Return 1 if *x* is less than or equal to *y*. Otherwise, return 0.
short **__mlt(** signed int *x* , signed int *y* **);**	**MCMP32** *dst* , *src*	Return 1 if *x* is less than *y*. Otherwise, return 0.
short **__mltu(** unsigned int *x* , unsigned int *y* **);**	**MCMP32** *dst* , *src*	Return 1 if *x* is less than *y*. Otherwise, return 0.
float **__mmaxf32(** float *x* , float *y* **);**	**MMAXF32** *dst* , *src*	Return the maximum of two 32-bit floating point values. If *src*>*dst*, copy src to dst.
float **__mminf32(** float *x* , float *y* **);**	**MMINF32** *dst* , *src*	Return the minimum of two 32-bit floating point values. If *src*<*dst*, copy src to dst.
__mnop()	**MNOP**	CLA no operation

Table 10-2. C/C++ Compiler Intrinsics for CLA (continued)

Intrinsic	Assembly Instruction(s)	Description
__msetflg(unsigned short *flag* **,** unsigned short *value* **)**	**MSETFLG** *flag* **,** *value*	Set/clear flag(s) in the MSTF register. The *flag* is a bit mask to indicate which bits to modify. The *value* provides the values to assign to these bits. See the *CLA Reference Guide* (SPRUGE6) for details about the MSETFLG instruction and the MSET register. This example sets the RNDF32 flag (bit 7) to 0, the TF flag (bit 6) to 0, and the NF flag (bit 2) to 1. The 0b prefix is a GCC language extension indicating that these are binary numbers. `__msetflg(0b11000100, 0b00000100);`
void **__mswapf(**float &*a* **,** float &*b* **);**	**MSWAPF** *a* **,** *b*	Swap the contents of *a* and *b*.
float **__sqrt(**float *x* **);**	**MEISQRTF32** *x*	Estimate the square root of 1/*x* to about 8 bits of precision. The Newton-Raphson method is used to approximate the square root. This is an alias for the __meisqrtf32 intrinsic.
float **__u32_bits_as_f32(** uint32_t *src*);	--	Packs a 32-bit register as a float. This intrinsic generates no code; it tells the compiler to change how it interprets the bits in place. See Section 7.6.1 for examples.
double **__u64_bits_as_f64(** uint64_t *src*);	--	Packs a 64-bit register as a double. This intrinsic generates no code; it tells the compiler to change how it interprets the bits in place. See Section 7.6.1 for examples.

Comparison intrinsics, such as __mlt and __mleq are provided for CLA to avoid possible overflows when performing signed integer comparisons.

The following intrinsics are available only within CLA2 background tasks. The background task has the lowest priority task and so will always be interrupted when another interrupt is available. These intrinsics can be used to temporarily prevent such interrupts from suspending the background task.

- __disable_interrupts();
- __enable_interrupts();

10.2.3 C Language Restrictions

There are several additional restrictions to the C language for CLA.

- Defining and initializing global/static data is not supported.

  Since the CLA code is executed in an interrupt driven environment, there is no C system boot sequence. As a result, global/static data initialization must be done during program execution, either by the C28x driver code or within a CLA function.

  Variables defined as const can be initialized globally. The compiler creates initialized data sections named .const_cla to hold these variables.
- CLA code cannot call C28x functions. The linker provides a diagnostic message if code compiled for C28 calls code compiled for CLA or if code compiled for CLA calls code compiled for C28.
- Recursive function calls are not supported.
- The use of function pointers is not supported.

Most GCC extensions are supported by the CLA compiler. Both the C and the CLA compiler have GCC extensions enabled by default. See the --relaxed_ansi option in Section 6.14.3 and the list of GCC language extensions in Section 6.15 for more information.

The interrupt function attribute described in Section 6.9.15 can be used with CLA interrupts, so long as you do not include any arguments to the interrupt attribute. The INTERRUPT pragma is also supported by the CLA compiler. For example, the following uses are supported:

```
__attribute__((interrupt))
void interrupt_name(void) {...}
#pragma INTERRUPT(interrupt_name);
void interrupt _name(void) {...}
```

For CLA2 background tasks, the "BACKGROUND" argument specifies that this is a background task instead of a regular interrupt. The following uses are supported.

```
__attribute__((interrupt("BACKGROUND")))
 void task_name(void) {...}
#pragma INTERRUPT(task_name, "BACKGROUND");
void task_name(void) {...}
```

The CLA compiler does not permit function calls to be made in background tasks. Inline functions are permitted and can be used instead.

10.2.4 Memory Model - Sections

Uninitialized global data is placed in section .bss_cla.

Initialized const data is placed in section .const_cla.

There is no C system heap for CLA, because there is no support for malloc().

Local variables and compiler temps are placed into a scratchpad memory area, which acts as the CLA C software stack. It is expected that the scratchpad memory area is defined and managed in the application's linker command file.

Instead of using a stack, each function has a generated function frame that is part of the .scratchpad section. Therefore, the only section that needs to be placed in the linker command file is the .scratchpad section. A scratchpad frame is designated for each function to hold local data, function arguments, and temporary storage. The linker determines which function frames can be overlaid in placement to save memory.

CLA function frames are placed in the .scratchpad section and are named in the form ".scratchpad:*functionSectionName*". Each function has its own subsection, and thus a unique section name. For example, if the source-level function name is "Cla1Task1", then the COFF function name will be "_Cla1Task1", the function section name will be "Cla1Prog:_Cla1Task1", and the function's scratchpad frame will be named ".scratchpad:Cla1Prog:_Cla1Task1". The function's scratchpad frame will use the base symbol "__cla_Cla1Task1_sp".

CLA2 background tasks are placed in the .scratchpad section and are named in the form ".scratchpad:background:*functionSectionName*". The background task frame cannot be overlaid with any other function frames, since the background task is likely to be returned to after yielding to interrupts.

Note that if an assembly writer uses a different naming convention for the data space of a function, it cannot be overlaid nor placed within the .scratchpad section.

It is not necessary to specify a size for the .scratchpad section.

CLA object files compiled with compiler versions prior to 6.4 are compatible with newly generated object files so long as the linker command file supports both scratchpad naming conventions. However, the scratchpad section used for old object files cannot be overlaid with the new .scratchpad section and you must ensure that enough memory is available for both sections.

10.2.5 Function Structure and Calling Conventions

The CLA compiler supports multiple nested levels of function calls. The CLA compiler also supports calling functions with more than two arguments.

Pointer arguments are passed in MAR0 and MAR1. 32-bit values are passed in MR0, MR1, and MR2. 16-bit values are passed in MR0, MR1, and MR2. Any further arguments are passed on the function frame (function-local scratchpad space), starting at offset 0.

All registers except for MR3 are saved on call. MR3 is saved on entry.

When interfacing with CLA assembly language modules, use these calling conventions to interface with compiled CLA code.

Appendix A

Glossary

A.1 Terminology

absolute lister — A debugging tool that allows you to create assembler listings that contain absolute addresses.

alias disambiguation — A technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

aliasing — The ability for a single object to be accessed in more than one way, such as when two pointers point to a single object. It can disrupt optimization, because any indirect reference could refer to any other object.

allocation — A process in which the linker calculates the final memory addresses of output sections.

ANSI — American National Standards Institute; an organization that establishes standards voluntarily followed by industries.

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.

byte Per ANSI/ISO C, the smallest addressable unit that can hold a character. For TMS320C28x, the size of a byte is 16-bits, which is also the size of a word.

C/C++ compiler A software program that translates C source statements into assembly language source statements.

code generator A compiler tool that takes the file produced by the parser or the optimizer and produces an assembly language source file.

COFF Common object file format; a system of object files configured according to a standard developed by AT&T. These files are relocatable in memory space.

command file A file that contains options, filenames, directives, or commands for the linker or hex conversion utility.

comment A source statement (or portion of a source statement) that documents or improves readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

compiler program A utility that lets you compile, assemble, and optionally link in one step. The compiler runs one or more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

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 TMS320C28x 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.

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.

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.

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.
overlay page	A section of physical memory that is mapped into the same address range as another section of memory. A hardware switch determines which range is active.
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.
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.

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.

structure A collection of one or more variables grouped together under a single name.

subsection A relocatable block of code or data that ultimately will occupy continuous space in the memory map. Subsections are smaller sections within larger sections. Subsections give you tighter control of the memory map.

symbol A string of alphanumeric characters that represents an address or a value.

symbolic debugging The ability of a software tool to retain symbolic information that can be used by a debugging tool such as an emulator or simulator.

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 ^.

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.

unsigned value A value that is treated as a nonnegative number, regardless of its actual sign.

variable A symbol representing a quantity that can assume any of a set of values.

word A 16-bit addressable location in target memory

Changes from December 16, 2020 to June 15, 2021 (from Revision V (December 2020) to Revision W (June 2021)) **Page**

Changes from September 1, 2020 to December 15, 2020 (from Revision U (September 2020) to Revision V (December 2020)) **Page**

Changes from February 28, 2020 to August 31, 2020 (from Revision T (February 2020) to Revision U (August 2020)) **Page**

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.

Earlier Revisions

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPRU514T	Using the Compiler	Section 2.3.4	Inlined memcpy calls now support >255 words through the use of RPT with a register operand. This allows inlining of memcpy of up to 65535 words. However, the maximum value specified with --rpt_threshold is still 256.
SPRU514T	Using the Compiler, Run-Time Environment	Section 2.3.4 and Section 7.6	Clarified that TMU1 support is available for EABI only.
SPRU514T	Using the Compiler, CLA Compiler	Section 2.3, Section 10.1.1, and Section 10.2.2	Added the --cla_signed_compare_workaround option for the CLA compiler. Added comparison intrinsics for the CLA compiler.
SPRU514T	Linking	Section 4.3.5	Clarified that either --rom_model or --ram_model is required if only the linker is being run, but --rom_model is the default if the compiler runs on C/C++ files on the same command line.
SPRU514T	C/C++ Language	Section 6.9.21	The `#pragma once` is now documented for use in header files.
SPRU514T	C/C++ Language	Section 6.15.4, Section 10.2.1, and Section 10.2.3	The linker now provides diagnostics about certain problems with blocking access and interactions between CLA and C28 code.
SPRU514T	Run-Time Environment	Section 7.6.3	Corrected assembly instruction equivalents for __atan, __cos, and __sin TMU intrinsics.
SPRU514T	Run-Time Environment	Section 7.6.4	Clarified that fast integer division requires EABI and FPU32 or FPU64. The types returned by fast integer division intrinsics that divide a 64-bit value by a 32-bit value have changed. In addition, the assembly performed by these intrinsics has been optimized.
SPRU514T	Run-Time Environment	Section 7.10.4.1	Clarified that zero initialization takes place only if the --rom_model linker option is used, not if the --ram_model option is used.
SPRU514S		-- throughout --	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.
SPRU514S	Using the Compiler, Optimization	Section 2.3 and Section 3.14	The --isr_save_vcu_regs compiler option has been added.
SPRU514S	CLA Compiler	Section 10.2.1 and Section 10.2.3	Added information about declaring variables shared by CLA and C28x code and about function calls between C28x and CLA code.
SPRU514R.1	Using the Compiler, Run-Time Environment	Section 2.3.4, Section 7.6.4, and Section 7.8.2	Added information about fast integer division in regard to the built-in integer division and modulo operators ("/" and "%") and intrinsics.
SPRU514R.1	Run-Time Environment	Section 7.1.7	Clarify bit-field alignment rules.
SPRU514R.1	Run-Time Environment	Section 7.6.2	Added intrinsics specific to FPU64 and information about COFF vs. EABI use of FPU64.
SPRU514R.1	CLA Compiler	Section 10.2	Clarified recommendation for int type usage.
SPRU514R		-- throughout --	Added further documentation of EABI support. This includes identifying features that are supported only for COFF or EABI. Identified examples as COFF-specific where necessary.

Earlier Revisions (continued)

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPRU514R	Using the Compiler	Section 2.3	Added command-line options for EABI, including --cinit_compression, --copy_compression, --extern_c_can_throw, --retain, --unused_section_elimination, and --zero_init.
SPRU514R	Using the Compiler	Section 2.3.1	Added the --emit_references:file linker option.
SPRU514R	Using the Compiler	Section 2.3.4	Added the --silicon_errata_fpu1_workaround option.
SPRU514R	Using the Compiler	Section 2.5.1	Documented that C standard macros such as __STDC_VERSION__ are supported.
SPRU514R	Using the Compiler	Section 2.11.1	Added information about situations where intrinsics may not be inlined.
SPRU514R	Using the Compiler, C/C++ Language	Section 2.13, Section 6.11	Added information about supported Application Binary Interfaces (ABIs).
SPRU514R	Linking C/C++ Code, Run-Time Environment	Section 4.3.6, Section 7.1.1	Added EABI-specific sections such as .bss, .const, .sysmem, and .init_array. Also added .args, .ppdata, and .ppinfo sections.
SPRU514R	C/C++ Language	Section 6.3	Added EABI-specific information about double and wchar_t data types.
SPRU514R	C/C++ Language	Section 6.9	Added the LOCATION, NOINIT, PERSISTENT, and WEAK pragmas.
SPRU514R	C/C++ Language	Section 6.12	Added EABI-specific information about namespaces.
SPRU514R	C/C++ Language	Section 6.14.1	Updated list of C99 non-supported run-time functions.
SPRU514R	C/C++ Language	Section 6.15.2	Added documentation for the aligned, calls, naked, and weak function attributes.
SPRU514R	C/C++ Language	Section 6.15.4	Added documentation for the location, noinit, persistent, and weak variable attributes.
SPRU514R	Run-Time Environment	Section 7.2.1	Added FPU64 registers.
SPRU514R	Run-Time Environment	Section 7.6	Corrected descriptions of values returned by TMU intrinsics such as __sin() and __cos().
SPRU514R	Run-Time Environment	Section 7.10.4	Added information about automatic initialization of variables for EABI.
SPRU514R	Run-Time Support Functions	Section 8.1.8	Added EABI-specific run-time library naming conventions.
SPRU514R	Run-Time Support Functions	DEV_lseek topic	Corrected syntax documented for DEV_lseek function.
SPRU514Q	Introduction, Using the Compiler, C/C++ Language	Section 1.3, Section 2.3, Section 6.1, and Section 6.14.2	Added support for C11.
SPRU514Q	Using the Compiler	Section 2.3.4	Added support for EABI. The COFF ABI is the default.
SPRU514Q	Using the Compiler	Section 2.3.1	Added the --ecc=on linker option, which enables ECC generation. Note that ECC generation is now off by default.
SPRU514Q	Using the Compiler	Section 2.3.4 and Section 7.6	Added 64-bit FPU support via --float_support=fpu64.
SPRU514Q	Using the Compiler, Run-Time Environment	Section 2.3.4 and Section 7.6	Added support for fast integer division via --idiv_support.
SPRU514Q	Using the Compiler, C/C++ Language	Section 2.3.4 and Section 7.6	Added further TMU support via --tmu_support=tmu1.
SPRU514Q	Using the Compiler	Section 2.3.4	Added Cyclic Redundancy Check support via --vcu_support=vcrc.
SPRU514Q	Using the Compiler	Section 2.5.1	The __TI_STRICT_ANSI_MODE__ and __TI_STRICT_FP_MODE__ macros are defined as 0 if their conditions are false.
SPRU514Q	Using the Compiler, C/C++ Language	Section 2.11 and Section 6.9	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.

Earlier Revisions (continued)

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPRU514Q	Optimization, C/C++ Language	Section 3.11 and Section 6.15.4	Added blocked and noblocked attributes for better Data Page (DP) pointer load optimization. The --disable_dp_load_opt option is no longer recommended.
SPRU514Q	C/C++ Language	Section 6.2	Removed several C++ features from the exception list because they have been supported for several releases.
SPRU514Q	C/C++ Language	Section 6.3.3 and Section 7.6	Added a recommendation that 32-bit floating point values be declared as float, not as double. (Both are currently 32 bits.) Modified intrinsic syntax descriptions to use "float" for 32-bit values.
SPRU514Q	C/C++ Language	Section 6.4	Added information about character sets and file encoding.
SPRU514Q	C/C++ Language	Section 6.15.2 and Section 6.15.4	Added "retain" as a function attribute and variable attribute.
SPRU514Q	C/C++ Language	Section 6.15.6	Clarified the availability of the __builtin_sqrt() and __builtin_sqrtf() functions.
SPRU514Q	CLA Compiler	Section 10.2	Corrected the syntax for the __mswapf intrinsic.
SPRU514P	C/C++ Language	Section 6.15	The compiler now supports several Clang __has_ macro extensions.
SPRU514P	C/C++ Language	Section 6.15.1	The wrapper header file GCC extension (#include_next) is now supported.
SPRU514O	C/C++ Language	Section 6.5.1	Clarified exceptions to const data storage set by the const keyword.
SPRU514N	Optimization	Section 3.7.1.4	Corrected error in command to process the profile data.
SPRU514M	Using the Compiler C/C++ Language, and CLA Compiler	Section 2.3, Section 2.3.4, Section 2.5.1, Section 6.9.15, and Section 10.2	Documented support for CLA version 2 and CLA v2 background tasks.
SPRU514M	Using the Compiler, C/C++ Language	Section 2.3.3	Revised to state that --check_misra option is required even if the CHECK_MISRA pragma is used.
SPRU514M	Using the Compiler	Section 2.3.5	Removed the --symdebug:coff option, which is no longer supported.
SPRU514M	Using the Compiler	Section 2.10	Corrected the document to describe the ---gen_preprocessor_listing option. The name --gen_parser_listing was incorrect.
SPRU514M	Optimization	Section 3.11	Provided information about data page blocking.
SPRU514L	Optimization	Section 3.7.3	Corrected function names for _TI_start_pprof_collection() and _TI_stop_pprof_collection().
SPRU514L	CLA Compiler	Section 10.2	Provided additional information and an example for the __msetflg intrinsic.
SPRU514K	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.
SPRU514J	Using the Compiler	Section 2.3 and Section 4.2.2	The --gen_data_subsections option has been added.
SPRU514J	Using the Compiler	Section 2.3.5	The --symdebug:dwarf_version compiler option has been added. This option sets the DWARF debugging format version used.
SPRU514J	Optimization	Section 3.7 and Section 3.8	Feedback directed optimization is described. This technique can be used for code coverage analysis.
SPRU514J	C/C++ Language	Section 6.9.1	A CALLS pragma has been added to specify a set of functions that can be called indirectly from a specified calling function. Using this pragma allows such indirect calls to be included in the calculation of a functions' inclusive stack size.
SPRU514J	C/C++ Language	Section 6.15.7	A byte_peripheral type attribute and an intrinsic have been added to access byte peripheral data.
SPRU514J	Run-Time Environment	Section 7.6	Intrinsics have been added to perform unsigned integer division. The new intrinsics are __euclidean_div_i32byu32(), __rpt_subcul(), and __subcul().
SPRU514J	Run-Time Environment	Section 7.10.1	Additional boot hook functions are available. These can be customized for use during system initialization.
SPRU514I	Using the Compiler	Table 2-7	The --cla_default option has been added. This option causes files with an extension of .c to be processed as CLA files.

Earlier Revisions (continued)

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPRU514I	Using the Compiler	Section 2.3.4	The --ramfunc option has been added. If set, this option places all functions in RAM.
SPRU514I	Using the Compiler	--	The --no_fast_branch option has been deprecated.
SPRU514I	C/C++ Language	Section 6.14.1	C99 math support is now available, including float and long double versions of floating point math functions.
SPRU514I	C/C++ Language	Section 6.15.2	The ramfunc function attribute has been added. It specifies that a function should be placed in RAM.
SPRU514I	Run-Time Environment	Section 7.3.2	Added XAR6 to the list of registers and corrected the location in which the address of the returned structure is placed.
SPRU514I	Run-Time Environment	Section 7.6	Added the __eallow and __edis intrinsics.
SPRU514I	CLA Compiler	Section 10.2.3	Most GCC extensions are now supported by the CLA compiler.
SPRU514H	Introduction	Section 1.3	Added support for C99 and C++03.
SPRU514H	Using the Compiler	Table 2-7	Added support for C99 and C++03. The -gcc option has been deprecated. The --relaxed_ansi option is now the default.
SPRU514H	Using the Compiler	Section 2.3.3	Added the --advice:performance option.
SPRU514H	Using the Compiler	Section 2.3.4	The --silicon_version=27 option is no longer supported.
SPRU514G	Using the Compiler	Section 2.3.4	Added --tmu_support=tmu0 option. This option also affects the behavior of the --float_support and --fp_mode=relaxed options.
SPRU514G	Using the Compiler	Section 2.3.4	Added support for Type 1 CLA via --cla_support=cla1.
SPRU514G	Using the Compiler	Section 2.3.4	Added support for Type 2 VCU via --vcu_support=vcu2.
SPRU514H	Using the Compiler	Section 2.3.11	Added information about the --flash_prefetch_warn option.
SPRU514H	Using the Compiler	Section 2.5.1	Added several predefined macro names that were not documented.
SPRU514H	Using the Compiler	Section 2.5.3	Documented that the #warning and #warn preprocessor directives are supported.
SPRU514H	Using the Compiler	Section 2.6	Added section on techniques for passing arguments to main().
SPRU514H	Using the Compiler	Section 2.11	Documented that the inline keyword is now enabled in all modes except C89 strict ANSI mode.
SPRU514H	C/C++ Language	Section 6.3	The size of pointer types on C28x is now 32 bits instead of 22 bits. The near and far keywords are deprecated. The small memory model is no longer supported; the only memory model uses 32-bit pointers. The .bss, .const, and .sysmem sections are no longer used; the .ebss, .econst, and .esysmem sections are used instead. (Symbol addresses are assumed to be less than 22 bits for performance reasons.)
SPRU514H	C/C++ Language	Section 6.1.1	Added section documenting implementation-defined behavior.
SPRU514H	C/C++ Language	Section 6.3.1	Added documentation on the size of enum types.
SPRU514H	C/C++ Language	Section 6.9.15, Section 6.9.22, and Section 6.15.2	Added C++ syntax for the INTERRUPT and RETAIN pragmas. Also removed unnecessary semicolons from #pragma syntax specifications. Also the GCC interrupt and alias function attributes are now supported.
SPRU514H	C/C++ Language	Section 6.9.11 and Section 6.9.12	Added the FUNC_ALWAYS_INLINE and FUNC_CANNOT_INLINE pragmas.
SPRU514H	C/C++ Language	Section 6.9.7	Added the diag_push and diag_pop diagnostic message pragmas.
SPRU514H	C/C++ Language	Section 6.14, Section 6.14.1, and Section 6.14.3	Added support for C99 and C++03. The --relaxed_ansi option is now the default and --strict_ansi is the other option; "normal mode" for standards violation strictness is no longer available.
SPRU514H	Run-Time Environment	Section 7.4	Added reference to section on accessing linker symbols in C and C++ in the *Assembly Language Tools User's Guide*.
SPRU514G	Run-Time Environment	Table 7-6 and Table 7-8	Added intrinsics for TMU instructions and for reading from and writing to memory using 32-bit addresses for data placed higher than the usual 22-bit address range.
SPRU514H	Run-Time Support Functions	Section 8.1.3	RTS source code is no longer provided in a rtssrc.zip file. Instead, it is located in separate files in the lib/src subdirectory of the compiler installation.

Earlier Revisions (continued)

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPRU514H	C++ Name Demangler	Section 9.1	Corrected information about name demangler options.
SPRU514H	CLA Compiler	Section 10.1	Non-recursive function calls and more than two function parameters are now supported for CLA code. CLA scratchpad management has been simplified; it is no longer necessary to specify a size for the scratchpad in the linker command file. The compiler now supports both the interrupt attribute and the INTERRUPT pragma for CLA interrupts.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), 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, 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 (https:www.ti.com/legal/termsofsale.html) 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.

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