TMS320C55x Optimizing C/C++ Compiler v 4.4

User's Guide

TEXAS INSTRUMENTS

Literature Number: SPRU281G
December 2011

Contents

List of Figures

List of Tables

Read This First

About This Manual

The *TMS320C55x Optimizing C/C++ Compiler User's Guide* explains how to use these compiler tools:

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

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

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

Notational Conventions

This document uses the following conventions:

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

  Here is a sample of C code:

  ```
  #include <stdio.h>
  main()
  {    printf("hello, cruel world\n");
  }
  ```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** and parameters are in an *italic typeface*. Portions of a syntax that are in bold should be entered as shown; portions of a syntax that are in italics describe the type of information that should be entered.
- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in the **bold typeface**, do not enter the brackets themselves. The following is an example of a command that has an optional parameter:

cl55 [*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:

cl55 --run_linker {**--rom_model | --ram_model**} *filenames* [**--output_file=** *name.out*]
--library= *libraryname*

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

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

- Some directives can have a varying number of parameters. For example, the .byte directive. This syntax is shown as [, ..., *parameter*].
- The TMS320C55x™ device is referred to as C55x.

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 C Standard), International Organization for Standardization

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

Related Documentation From Texas Instruments

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

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

SPRU280: —TMS320C55x Assembly Language Tools User's Guide. Describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for TMS320C55x devices.

SPRU317: —TMS320C55x DSP Peripherals Reference Guide. Introduces the peripherals, interfaces, and related hardware that are available on TMS320C55x DSPs.

SPRU376— TMS320C55x DSP Programmer's Guide. Describes ways to optimize C and assembly code for the TMS320C55x DSPs and explains how to write code that uses special features and instructions of the DSP.

SPRU393: —TMS320C55x Technical Overview. Introduces the TMS320C55x digital signal processor (DSP). The TMS320C55x is the latest generation of fixed-point DSPs in the TMS320C5000™ DSP platform. Like the previous generations, this processor is optimized for high performance and low-power operation. This book describes the CPU architecture, low-power enhancements, and embedded emulation features of the TMS320C55x.

SWPU067: —TMS320C55x 3.0 DSP Mnemonic Instruction Set Reference Guide. Describes the TMS320C55x 3.0 DSP mnemonic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the mnemonic instruction set.

SWPU068: —TMS320C55x 3.0 DSP Algebraic Instruction Set Reference Guide. Describes the TMS320C55x 3.0 DSP algebraic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the mnemonic instruction set.

SWPU073: —TMS320C55x DSP CPU Reference Guide, Version 3.0. Describes the architecture, registers, and operation of the CPU for the TMS320C55x digital signal processors (DSPs).

TMS320C55x, XDS510E are trademarks of Texas Instruments.
All other trademarks are the property of their respective owners.

Chapter 1

SPRU281G–December 2011

Introduction to the Software Development Tools

The TMS320C55x™ is supported by a set of software development tools, which includes an optimizing C/C++ compiler, an assembler, a linker, and assorted utilities.

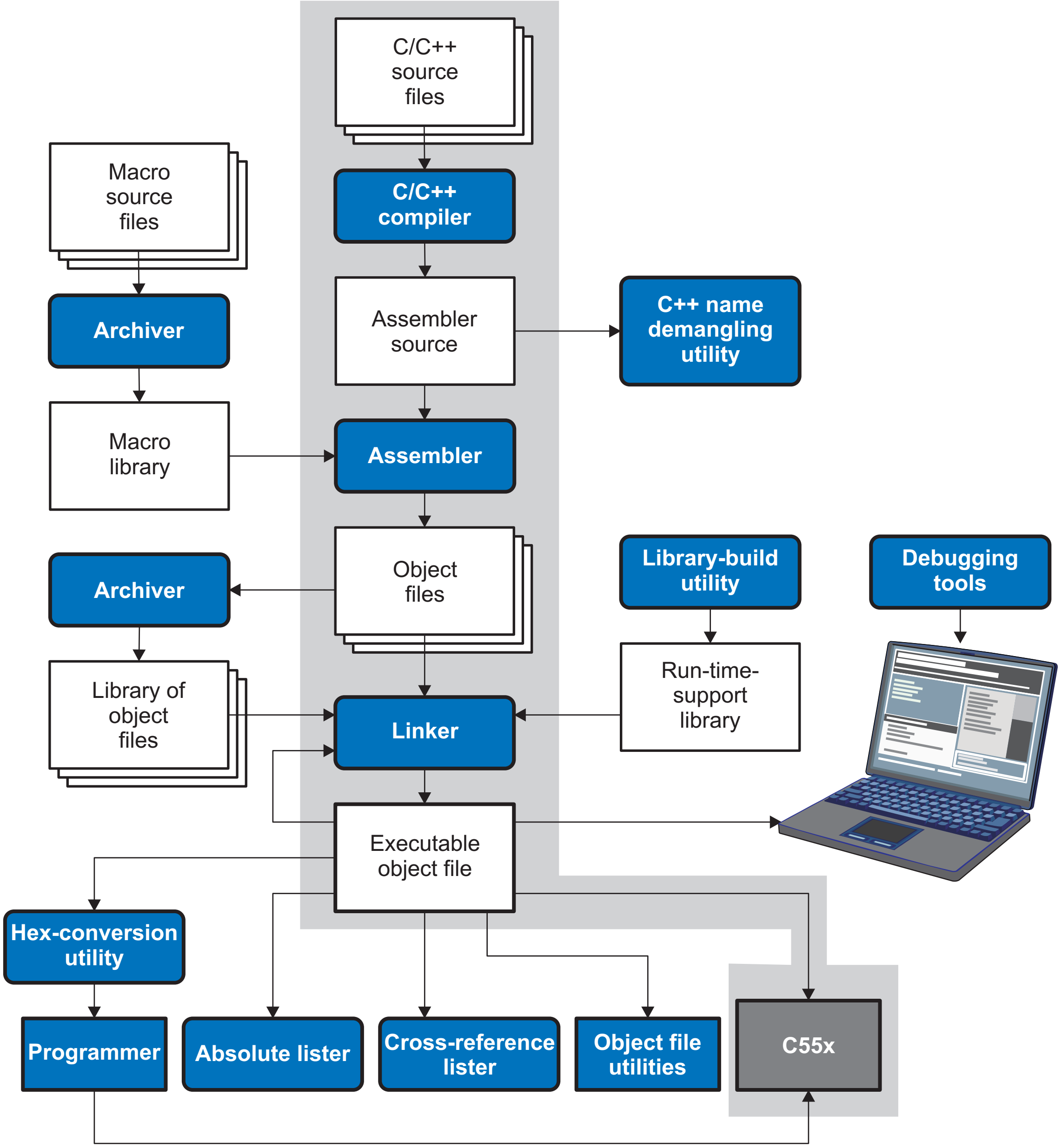
This chapter provides an overview of these tools and introduces the features of the optimizing C/C++ compiler. The assembler and linker are discussed in detail in the *TMS320C55x Assembly Language Tools User's Guide*.

Topic **Page**

1.1 Software Development Tools Overview

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

Figure 1-1. TMS320C55x 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 C55x assembly language source code. See Chapter 2.
- The **assembler** translates assembly language source files into machine language relocatable object files. The TMS320C55x assembler accepts algebraic and mnemonic (including C54x mnemonic) source formats. The *TMS320C55x Assembly Language Tools User's Guide* explains how to use the assembler.

- 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. The *TMS320C55x Assembly Language Tools User's Guide* provides a complete description of the linker.
- The **archiver** allows you to collect a group of files into a single archive file, called a *library*. Additionally, the archiver allows you to modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object files. The *TMS320C55x Assembly Language Tools User's Guide* explains how to use the archiver.
- 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 7.

  You can use the **library-build utility** to build your own customized run-time-support library. See Section 7.4. Source code for the standard run-time-support library functions for C and C++ are provided in the self-contained rtssrc.zip file.
- The **hex conversion utility** converts an object file into other object formats. You can download the converted file to an EPROM programmer. The *TMS320C55x Assembly Language Tools User's Guide* explains how to use the hex conversion utility and describes all supported formats.
- The **absolute lister** accepts linked object files as input and creates .abs files as output. You can assemble these .abs files to produce a listing that contains absolute, rather than relative, addresses. Without the absolute lister, producing such a listing would be tedious and would require many manual operations. The *TMS320C55x Assembly Language Tools User's Guide* explains how to use the absolute lister.
- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files. The *TMS320C55x Assembly Language Tools User's Guide* explains how to use the cross-reference utility.
- The **C++ name demangler** is a debugging aid that converts names mangled by the compiler back to their original names as declared in the C++ source code. As shown in Figure 1-1, you can use the C++ name demangler on the assembly file that is output by the compiler; you can also use this utility on the assembler listing file and the linker map file. See Chapter 8.
- The **disassembler** decodes object files to show the assembly instructions that they represent. The *TMS320C55x Assembly Language Tools User's Guide* explains how to use the disassembler.
- The main product of this development process is an executable object file that can be executed in a **TMS320C55x** device. You can use one of several debugging tools to refine and correct your code. Available products include:
  - An instruction-level software simulator
  - An extended development system (XDS510E™) emulator

1.2 C/C++ Compiler Overview

The following subsections describe the key features of the compiler.

1.2.1 ANSI/ISO Standard

The C and C++ language features in the compiler are implemented in conformance with these ISO standards:

- **ISO-standard C**

  The C/C++ compiler conforms to the C Standard ISO/IEC 9889:1990. The ISO standard supercedes and is the same as the ANSI C standard. There is also a 1999 version of the ISO standard, but the TI compiler conforms to the 1990 standard, not the 1999 standard. The language is also described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R).
- **ISO-standard C++**

  The C/C++ compiler conforms to the C++ Standard ISO/IEC 14882:1998. The language is also described in Ellis and Stroustrup's *The Annotated C++ Reference Manual* (ARM), but this is not the standard. The compiler also supports embedded C++. For a description of *unsupported* C++ features, see Section 5.2.

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

  The compiler tools come with an extensive run-time library. All library functions conform to the ISO C/C++ library standard. The library includes functions for standard input and output, string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, and exponential and hyperbolic functions. Functions for signal handling are not included, because these are target-system specific. For more information, see Chapter 7.

1.2.2 Output Files

These output files are created by the compiler:

- **COFF object files**

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

1.2.3 Compiler Interface

These features enable interfacing with the compiler:

- **Compiler program**

  The compiler tools include a compiler program (cl55) that you use to compile, optimize, assemble, and link programs in a single step. For more information, see Section 2.1

- **Flexible assembly language interface**

  The compiler has straightforward calling conventions, so you can write assembly and C functions that call each other. For more information, see Chapter 6.

1.2.4 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 7.4.

- **C++ name demangler**

  The C++ name demangler (dem55) 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 8.

- **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 files output by the compiler can be converted to EPROM programmer data files by using the hex conversion utility, as described in the *TMS320C55x Assembly Language Tools User's Guide*.

Using the C/C++ Compiler

The compiler translates your source program into machine language object code that the TMS320C55x can execute. Source code must be compiled, assembled, and linked to create an executable object file. All of these steps are executed at once by using the compiler.

Topic **Page**

2.1 About the Compiler

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

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

  You can compile C, C++, and assembly files in a single command. The compiler uses the filename extensions to distinguish between different file types. See Section 2.3.9 for more information.
- The **linker** combines object files to create an executable object file. The linker is optional, so you can compile and assemble many modules independently and link them later. See Chapter 4 for information about linking the files.

> **Invoking the Linker**
>
> **NOTE:** By default, the compiler does not invoke the linker. You can invoke the linker by using the --run_linker compiler option.

For a complete description of the assembler and the linker, see the *TMS320C55x Assembly Language Tools User's Guide*.

2.2 Invoking the C/C++ Compiler

To invoke the compiler, enter:

cl55 [*options*] [*filenames*] [**--run_linker** [*link_options*] *object files*]]

cl55 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 2-5 through Table 2-28.

filenames One or more C/C++ source files, assembly language source files, linear assembly files, or object files.

--run_linker Option that invokes the linker. The --run_linker option's short form is -z. See Chapter 4 for more information.

link_options Options that control the linking process.

object files Name of the additional object files for the linking process.

The arguments to the compiler are of three types:

- Compiler options
- Link options
- Filenames

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

```
cl55 symtab.c file.c seek.asm --run_linker --library=lnk.cmd
       --library=rts55.lib --output_file=myprogram.out
```

2.3 Changing the Compiler's Behavior With Options

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

For a help screen summary of the options, enter **cl55** with no parameters on the command line.

The following apply to the compiler options:

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

You can define default options for the compiler by using the C5X_C_OPTION environment variable. For a detailed description of the environment variable, see Section 2.4.1.

Table 2-5 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=*device*[:*revision*]	-v	Generates optimal code for the C55x device and optional revision number specified	Section 2.3.4
--memory_model={small| large | huge}	-ml	Specifies memory model. You can use -ml to alias --memory_model=large.	Section 2.3.3

Table 2-2. Optimization Options[1]

Option	Alias	Effect	Section
--opt_level=0	-O0	Optimizes register usage	Section 3.1
--opt_level=1	-O1	Uses -O0 optimizations and optimizes locally	Section 3.1
--opt_level=2	-O2 or -O	Uses -O1 optimizations and optimizes globally (default)	Section 3.1
--opt_level=3	-O3	Uses -O2 optimizations and optimizes the file	Section 3.1 Section 3.2
--opt_for_space=*n*	-ms	Controls code size on four levels (0, 1, 2, and 3)	Section 3.9

[1] **Note:** Machine-specific options (see Table 2-11) can also affect optimization.

Table 2-3. Debug Options

Option	Alias	Effect	Section
--symdebug:dwarf	-g	Enables symbolic debugging	Section 2.3.5 Section 3.8.1
--symdebug:coff		Enables symbolic debugging using the alternate STABS debugging format.	Section 2.3.5 Section 3.8.1
--symdebug:none		Disables all symbolic debugging	Section 2.3.5

Table 2-3. Debug Options (continued)

Option	Alias	Effect	Section
--symdebug:profile_coff		Enables profiling using the alternate STABS debugging format.	Section 2.3.5
--symdebug:skeletal		Enables minimal symbolic debugging that does not hinder optimizations (default behavior)	Section 2.3.5
--optimize_with_debug	-mn	Reenables optimizations disabled with --symdebug:dwarf	Section 3.8.1

Table 2-4. Include Options

Option	Alias	Effect	Section
--include_path=*directory*	`-I`	Defines #include search path	Section 2.5.2.1
--preinclude=*filename*		Includes *filename* at the beginning of compilation	Section 2.3.2

Table 2-5. 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 2.3.1
--run_linker	-z	Enables linking	Section 2.3.1
--skip_assembler	-n	Compiles or assembly optimizes only	Section 2.3.1

Table 2-6. Advanced Debug Options

Option	Alias	Effect	Section
--symdebug:keep_all_types		Keep unreferenced type information	Section 2.3.5

Table 2-7. Language Options

Option	Alias	Effect	Section
--cpp_default	-fg	Processes all source files with a C extension as C++ source files.	Section 2.3.7
--embedded_cpp	-pe	Enables embedded C++ mode	Section 5.13.3
--exceptions		Enables C++ exception handling	Section 5.6
--gcc		Enables support for GCC extensions	Section 5.14
--gen_acp_raw	-pl	Generates a raw listing file	Section 2.9
--gen_acp_xref	-px	Generates a cross-reference listing file	Section 2.8
--keep_unneeded_statics		Keeps unreferenced static variables.	Section 2.3.2
--kr_compatible	-pk	Allows K&R compatibility	Section 5.13.1
--multibyte_chars	-pc	Enables support for multibyte character sequences in comments, string literals and character constants.	--
--no_inlining	-pi	Disables definition-controlled inlining (but --opt_level=3 (or -O3) optimizations still perform automatic inlining)	Section 2.10
--no_intrinsics	-pn	Disables intrinsic functions. No predefinition of compiler-supplied intrinsic functions.	--
--program_level_compile	-pm	Combines source files to perform program-level optimization	Section 3.3
--relaxed_ansi	-pr	Enables relaxed mode; ignores strict ISO violations	Section 5.13.2
--rtti	-rtti	Enables run time type information (RTTI)	–-
--static_template_instantiation		Instantiate all template entities with internal linkage	–-
--strict_ansi	-ps	Enables strict ISO mode (for C/C++, not K&R C)	Section 5.13.2

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 2.5.7
--preproc_includes[=*filename*]	-ppi	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive	Section 2.5.8
--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 2.5.9
--preproc_only	-ppo	Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension.	Section 2.5.3
--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 2.5.5
--preproc_with_compile	-ppa	Continues compilation after preprocessing	Section 2.5.4
--preproc_with_line	-ppl	Performs preprocessing only. Writes preprocessed output with line-control information (#line directives) to a file with the same name as the input but with a .pp extension.	Section 2.5.6

Table 2-9. Predefined Symbols Options

Option	Alias	Effect	Section
--define=*name*[=*def*]	-D	Predefines *name*	Section 2.3.1
--undefine=*name*	-U	Undefines *name*	Section 2.3.1

Table 2-10. Diagnostics Options

Option	Alias	Effect	Section
--compiler_revision		Prints out the compiler release revision and exits	--
--diag_error=*num*	-pdse	Categorizes the diagnostic identified by *num* as an error	Section 2.6.1
--diag_remark=*num*	-pdsr	Categorizes the diagnostic identified by *num* as a remark	Section 2.6.1
--diag_suppress=*num*	-pds	Suppresses the diagnostic identified by *num*	Section 2.6.1
--diag_warning=*num*	-pdsw	Categorizes the diagnostic identified by *num* as a warning	Section 2.6.1
--display_error_number	-pden	Displays a diagnostic's identifiers along with its text	Section 2.6.1
--emit_warnings_as_errors	-pdew	Treat warnings as errors	Section 2.6.1
--gen_aux_user_info	-b	Generate user information file (.aux)	--
--issue_remarks	-pdr	Issues remarks (nonserious warnings)	Section 2.6.1
--no_warnings	-pdw	Suppresses warning diagnostics (errors are still issued)	Section 2.6.1
--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 2.6.1
--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 diagnostics that display the original source with line-wrap	Section 2.6.1
--write_diagnostics_file	-pdf	Generates a diagnostics information file. Compiler only option.	Section 2.6.1

Table 2-11. Run-Time Model Options

Option	Alias	Effect	Section
--algebraic	-mg	Outputs algebraic assembly	Section 2.3.3
--align_functions		Aligns all functions on a 4-byte boundary	Section 2.3.3
--asm_source={algebraic| mnemonic}		Selects assembly source format. The default behavior is --asm_source=mnemonic.	Section 2.3.3
--assert_arms_set	-ata	Asserts that assembly code should be compiled as if the ARMS status bit will be set during execution.	Section 2.3.11
--assert_c54cm_set	-atl	Asserts that the C54CM status bit will be enabled during the execution of this source file	Section 2.3.11
--assert_cpl_set	-atc	Asserts that the CPL status bit will be enabled during the execution of this source file	Section 2.3.11
--assert_sst_zero	-att	Asserts that the SST status bit will be disabled during the execution of this source file	Section 2.3.11
--assume_bss_onchip	-mb	Specifies that all data memory resides on-chip	Section 2.3.3
--assume_smul_off		For C55x only, specifies compiler should use 0 as the presumed value of the SMUL status bit.	Section 6.3.2
--bus_conflict	-atb	Causes the assembler to treat parallel bus conflict errors as warnings	Section 2.3.11
--calling_convention=*value*	-call	Forces compiler compatibility with specific calling conventions.	Section 2.3.3
--check_32bit_int_portability		Issues warning if compiler detects a source language construct that may not be portable	Section 2.3.3
--const_in_cinit	-mc	Allows constants normally placed in a .const section to be treated as read-only, initialized static variables	Section 2.3.3
--fp_reassoc={on|off}		Enables or disables the reassociation of floating-point arithmetic	Section 2.3.3
--gen_func_subsections={on|off}		Puts each function in a separate subsection in the object file	Section 4.2.2
--no_bad_aliases	-mt	Allows certain assumptions about aliasing and loops	Section 3.4.2
--no_block_repeat	-mr	Prevents the compiler from generating hardware blockrepeat, localrepeat, and repeat instructions	Section 2.3.3
--no_byte_operands		Prevents the compiler from generating low_byte and high_byte operands	--
--no_byte_io_operands		Prevents the compiler from generating low_byte and high_byte operand access to IO space	--
--no_mac_expand	--nomacx	Prevents the expansion of macros when source is output	Section 2.3.3
--no_nops_in_delay	-atn	Removes NOPs located in the delay slots of C54x delayed branch/call instructions	Section 2.3.3
--port_for_speed	-ath	Causes the assembler to encode C54x instructions for speed over size	Section 2.3.11
--predication_level=*limit*		Specifies maximum length of an instruction sequence on which to use conditional execution for each instruction.	Section 2.3.3
--profile:power		Enables power profiling	Section 2.3.5 Section 3.8.2
--ptrdiff_size={16|32}		Sets ptrdiff_t to an int (16 bits) or long (32 bits)	Section 2.3.3
--sat_reassoc={on|off}		Enables or disables the reassociation of saturating arithmetic. Default is --sat_reassoc=off.	Section 2.3.2
--small_enum		Uses the smallest possible size for the enumeration type	Section 2.3.3
--translate_c54x_mnem_source		Allows C54x native mnemonic assembly source	Section 2.3.11
--use_long_branch	-atv	Causes the assembler to use the largest form of certain variable-length instructions	Section 2.3.11

Table 2-12. 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 3.6
--call_assumptions=0	-op0	Specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler	Section 3.3.1
--call_assumptions=1	-op1	Specifies that the module contains variables modified from outside the source code provided to the compiler but does not use functions called from outside the source code	Section 3.3.1
--call_assumptions=2	-op2	Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler (default)	Section 3.3.1
--call_assumptions=3	-op3	Specifies that the module contains functions that are called from outside the source code provided to the compiler but does not use variables modified from outside the source code	Section 3.3.1
--gen_opt_info=0	-on0	Disables the optimization information file	Section 3.2.2
--gen_opt_info=1	-on1	Produces an optimization information file	Section 3.2.2
--gen_opt_info=2	-on2	Produces a verbose optimization information file	Section 3.2.2
--opt_for_space[=*n*]	-ms	Controls code size on four levels (0, 1, 2, and 3) (Default is 3.)	Section 3.9
--opt_for_speed[=*n*]	-mf	Controls speed over space (0-5 range) (Default is 4.)	Section 3.9
--optimizer_interlist	-os	Interlists optimizer comments with assembly statements	Section 3.7
--remove_hooks_when_inlining		Removes entry/exit hooks for auto-inlined functions	Section 2.12
--single_inline		Inlines functions that are only called once	--
--aliased_variables	-ma	Indicates that a specific aliasing technique is used	Section 3.4.1

[1] **Note:** Machine-specific options (see Table 2-11) can also affect optimization.

Table 2-13. Entry/Exit Hook Options

Option	Alias	Effect	Section
--entry_hook[=*name*]		Enables entry hooks	Section 2.12
--entry_parm={none|name|address}		Specifies the parameters to the function to the --entry_hook option	Section 2.12
--exit_hook[=*name*]		Enables exit hooks	Section 2.12
--exit_parm={none|name|address}		Specifies the parameters to the function to the --exit_hook option	Section 2.12

Table 2-14. Library Function Assumptions Options

Option	Alias	Effect	Section
--printf_support={nofloat|full|minimal}		Enables support for smaller, limited versions of the printf and sprintf run-time-support functions.	Section 2.3.2
--std_lib_func_defined	-ol1 or -oL1	Informs the optimizer that your file declares a standard library function	Section 3.2.1
--std_lib_func_not_defined	-ol2 or -oL2	Informs the optimizer that your file does not declare or alter library functions. Overrides the -ol0 and -ol1 options (default).	Section 3.2.1
--std_lib_func_redefined	-ol0 or -oL0	Informs the optimizer that your file alters a standard library function	Section 3.2.1

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 2.11 Section 3.7
--src_interlist	-s	Interlists optimizer comments (if available) and assembly source statements; otherwise interlists C and assembly source statements	Section 2.3.1
--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
--asm_undefine=*name*	-au	Undefines the predefined constant *name*	Section 2.3.11
--copy_file=*filename*	-ahc	Copies the specified file for the assembly module	Section 2.3.11
--cross_reference	-ax	Generates the cross-reference file	Section 2.3.11
--hash_optional_shift_count	-ats	(Mnemonic assembly only). Makes the # on literal shift counts optional.	Section 2.3.11
--include_file=*filename*	-ahi	Includes the specified file for the assembly module	Section 2.3.11
--no_const_clink		Stops generation of .clink directives for const global arrays.	Section 2.3.2
--output_all_syms	-as	Puts labels in the symbol table	Section 2.3.11
--purecirc		Informs the assembler that only C54x-specific circular addressing is used	Section 2.3.11
--suppress_asm_warnings	-atw	(Algebraic assembler only). Suppresses assembler warning messages	Section 2.3.11
--suppress_remark=*num*	-ar=*num*	Suppresses the assembler remark identified by *num*	Section 2.3.11
--syms_ignore_case	-ac	Makes case insignificant in assembly source files	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.	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 .obj directory.	Section 2.3.10
--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 .obj 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 2.3.1

Table 2-20. MISRA-C:2004 Options

Option	Alias	Effect	Section
--check_misra[={all|required|advisory|none|*rulespec*}]		Enables checking of the specified MISRA-C:2004 rules. Default is all.	Section 2.3.2
--misra_advisory={error|warning|remark|suppress}		Sets the diagnostic severity for advisory MISRA-C:2004 rules	Section 2.3.2
--misra_required={error|warning|remark|suppress}		Sets the diagnostic severity for required MISRA-C:2004 rules	Section 2.3.2

The following tables list the linker options. See the *TMS320C55x Assembly Language Tools User's Guide* for details on these options.

Table 2-21. Linker Basic Options

Option	Alias	Description
--output_file=*file*	-o	Names the executable output file. The default filename is a.out.
--map_file=*file*	-m	Produces a map or listing of the input and output sections, including holes, and places the listing in *filename*
--stack_size=*size*	[-]-stack	Sets primary C system stack size to *size* bytes and defines a global symbol that specifies the stack size. Default = 1K bytes
--heap_size=*size*	[-]-heap	Sets heap size (for the dynamic memory allocation in C) to *size* bytes and defines a global symbol that specifies the heap size. Default = 2K bytes

Table 2-22. File Search Path Options

Option	Alias	Description
--library=*file*	-l	Names an archive library or link command *file* as linker input
--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.
--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
--disable_auto_rts		Disables the automatic selection of a run-time-support library

Table 2-23. Command File Preprocessing Options

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

Table 2-24. Diagnostic 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_warnings_as_errors	-pdew	Treat warnings as errors
--issue_remarks		Issues remarks (nonserious warnings)
--no_demangle		Disables demangling of symbol names in diagnostics
--no_warnings		Suppresses warning diagnostics (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 diagnostics that display the original source with line-wrap
--warn_sections	-w	Displays a message when an undefined output section is created

Table 2-25. Linker Output Options

Option	Alias	Description
--absolute_exe	-a	Produces an absolute, executable object file. This is the default; if neither --absolute_exe nor --relocatable is specified, the linker acts as if --absolute_exe were specified.
--generate_dead_funcs_list		Writes a list of the dead functions that were removed by the linker to file fname.
--mapfile_contents=*attribute*		Controls the information that appears in the map file.
--relocatable	-r	Produces a nonexecutable, relocatable output object file
--rom		Creates a ROM object
--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 2-26. Symbol Management Options

Option	Alias	Description
--entry_point=*symbol*	-e	Defines a global symbol that specifies the primary entry point for the executable object file
--globalize=*pattern*		Changes the symbol linkage to global for symbols that match *pattern*
--hide=*pattern*		Hides symbols that match the specified *pattern*
--localize=*pattern*		Make the symbols that match the specified *pattern* local
--make_global=*symbol*	-g	Makes *symbol* global (overrides -h)
--make_static	-h	Makes all global symbols static
--no_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
--scan_libraries	-scanlibs	Scans all libraries for duplicate symbol definitions
--symbol_map=*refname=defname*		Specifies a symbol mapping; references to the *refname* symbol are replaced with references to the *defname* symbol
--undef_sym=*symbol*	-u	Adds *symbol* to the symbol table as an unresolved symbol
--unhide=*pattern*		Excludes symbols that match the specified *pattern* from being hidden

Table 2-27. Run-Time Environment Options

Option	Alias	Description
--arg_size=*size*	--args	Reserve *size* bytes for the argc/argv memory area
--fill_value=*value*	-f	Sets default fill value for holes within output sections
--ram_model	-cr	Initializes variables at load time
--rom_model	-c	Autoinitializes variables at run time
--sys_stacksize	-sysstack	Sets the secondary system stack size to *size* bytes and defines a global symbol that specifies the secondary stack size. Default = 1K bytes

Table 2-28. Miscellaneous Options

Option	Alias	Description
--disable_clink	-j	Disables conditional linking of COFF object files
--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.

2.3.1 Frequently Used Options

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

--c_src_interlist Invokes the interlist feature, which interweaves original C/C++ source with compiler-generated assembly language. The interlisted C statements may appear to be out of sequence. You can use the interlist feature with the optimizer by combining the --optimizer_interlist and --c_src_interlist options. See Section 3.7. The --c_src_interlist option can have a negative performance and/or code size impact.

--cmd_file=*filename* Appends the contents of a file to the option set. You can use this option to avoid limitations on command line length or C style comments imposed by the host operating system. Use a # or ; at the beginning of a line in the command file to include comments. You can also include comments by delimiting them with /* and */. To specify options, surround hyphens with quotation marks. For example, "--"quiet.

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

```
cl55 --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 C5X_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 Code Composer Studio, enter the definition in a file and include that file with the --cmd_file option.

--gen_aux_user_info Generates a user information file and appends the .aux extension.

--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.

2.3.2 Miscellaneous Useful Options

Following are detailed descriptions of miscellaneous options:

--check_misra={all|required|advisory|none|*rulespec*} Displays the specified amount or type of MISRA-C documentation. The *rulespec* parameter is a comma-separated list of specifiers. See Section 5.3 for details.

--fp_reassoc={on|off} Enables or disables the reassociation of floating-point arithmetic. If --strict_ansi is set, --fp_reassoc=off is set since reassociation of floating-point arithmetic is an ANSI violation.

--keep_unneeded_statics Does not delete unreferenced static variables. The parser by default remarks about and then removes any unreferenced static variables. The --keep_unneeded_statics option keeps the parser from deleting unreferenced static variables and any static functions that are referenced by these variable definitions. Unreferenced static functions will still be removed.

--no_const_clink Tells the compiler to not generate .clink directives for const global arrays. By default, these arrays are placed in a .const subsection and conditionally linked.

--misra_advisory={error|warning|remark|suppress} Sets the diagnostic severity for advisory MISRA-C:2004 rules.

--misra_required={error|warning|remark|suppress} Sets the diagnostic severity for required MISRA-C:2004 rules.

--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 and sprintf run-time-support functions. The valid values are:

- full: Supports all format specifiers. This is the default.
- nofloat: Excludes support for printing floating point values. Supports all format specifiers except %f, %F, %g, %G, %e, and %E.
- minimal: Supports the printing 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.

2.3.3 Run-Time Model Options

These options are specific to the TMS320C55x toolset. See the referenced sections for more information. TMS320C55x-specific assembler options are listed in Section 2.3.11.

--algebraic

Causes the compiler and assembler to generate algebraic assembly.

--align_functions

Forces all functions in the compilation to be aligned on a 4-byte boundary. This enables program performance to remain consistent. Without this forced alignment, a change in one C function can lead to changed cycle counts in other (unchanged) C functions due to different alignment of branches and loops.

--asm_source={algebraic|mnemonic}

By default, the compiler and assembler use mnemonic assembly: the compiler generates mnemonic assembly output, and the assembler only accepts mnemonic assembly input files. This is equivalent to specifying --asm_source=mnemonic. When --asm_source=algebraic is specified, the compiler and assembler use algebraic assembly. You must use this option to assemble algebraic assembly input files, or to compile C code containing algebraic asm statements. Algebraic and mnemonic source code cannot be mixed in a single source file.

--assume_smul_off

For C55x, specifies that the compile should use 0 as the presumed value of the SMUL status bit (see Section 6.3.2).

--call=*value*

Forces compiler compatibility with specific calling conventions. Early versions of the C55x compiler used a calling convention that was less efficient. The new calling conventions address these inefficiencies. The --call option supports compatibility with existing code which either calls or is called by assembly code using the original convention. Using --call=c55_compat forces the compiler to generate code that is compatible with the original calling convention. Using --call=c55_new forces the compiler to use the new calling convention.

The compiler uses the new convention by default, except when compiling for P2 reserved mode, which sets the default to the original convention. Within a single executable, only one calling convention can be used. The linker enforces this rule.

--check_32bit_int_portability

Issues a warning if the compiler detects a source language construct that may not be portable when moved from a C environment where type int is represented in 32 bits to C55x C where int is 16 bits. Not all such portability issues can be detected at compile time.

--const_in_cinit

Allows constants that are normally placed in a .const section to be treated as read-only, initialized static variables. This is useful when using the small memory model or on P2 reserved mode hardware. It allows the constant values to be loaded into extended memory while the space used to hold the values at run time is still in the single data page used by the program.

--memory_model={huge|large|small}

Generates large or huge memory model code.

--no_block_repeat

Prevents the compiler from generating the hardware blockrepeat, localrepeat, and repeat instructions. This option is only useful when --opt_level=2 or --opt_level=3 is specified.

--predication_level=*number*

Tells the compiler the longest sequence of machine instructions that should be predicated using the conditional execute instruction. That instruction permits conditional execution to be performed without interrupting the hardware pipeline and thus results in significant cycle savings over a conditional branching code sequence. This sort of code generation results from source level conditional execution such as in an if-then or if-then-else construct.

By default the option value is 3, unless the --optimize for space option is used, in which case it is 2. The option can be set to a higher number to make the compiler more aggressive in performing this optimization. A value of 4 often will improve performance; higher values may in some cases improve performance, but only rarely. Normally this option is used to improve performance when a conditional construct, such as if-then, occurs inside a loop.

--prtrdiff_t_16

Changes the prtdiff_t type to an int (16 bits). The ptrdiff_t type, defined in the stddef.h or cstddef header, is a signed integer type that is the data type resulting from the subtraction of two pointers. This option ensures the ptrdiff_t type is an int.

-silicon_version=*device* [**:** *revision*]

Determines the processor for which instructions are generated. For information on legal values, see Section 2.3.4.

--small_enum

By default, the C55x compiler uses 16 bits for every enum. When you use the --small_enum option, the smallest possible byte size for the enumeration type is used. For example, enum example_enum {first = -128, second = 0, third = 127} uses only one byte instead of 16 bits when the --small_enum option is used. Do not link object files compiled with the --small_enum option with object files that have been compiled without it. If you use the --small_enum option, you must use it with all of your C/C++ files; otherwise, you will encounter errors that cannot be detected until run time.

--translate_c54x_mnem_source

Assembles mnemonic assembly language source that contains C54x assembly language and C55x assembly language as C55x object code. The --translate_c54x_mnem_source option is used in conjunction with the --asm_source=mnemonic option. (The default behavior is --asm_source=mnemonic so this option does not need to be specified.)

2.3.4 Selecting the Device Version (--silicon_version Option)

The --silicon_version (alias -v) option is used to indicate which target device or devices will be used to execute the code being generated. Using the -vdevice[:revision] option specifies the target for which instructions should be generated. The device parameter is usually the last four digits of the TMS320C55x part number. For example, to specify the TMS320VC5510 DSP device, you would use --silicon_version=5510. The --silicon_version=*device* option without a revision number generates code that will run on all current silicon revisions for that device. However, if you specify a revision number (e.g., --silicon_version=5510:1), the compiler may be able to generate more optimal code for that revision. You can use multiple --silicon_version options to specify multiple devices and revision numbers. Certain combinations may not be permitted.

CAUTION

Non-Use Versus Use of --silicon_version
If the --silicon_version option is not used, the compiler generates code that will run on all known TMS320C55x devices. However, code which will execute on all devices is not optimal for any device. Through careful use of --silicon_version, you avoid workarounds for, and detection of, hardware defects in devices that you are not using.

The --silicon_version=*device*:0 option generates code according to the device's original hardware specification.

The revision specifier can be more than one digit, if necessary. A revision numbered as 1.1 would be specified as --silicon_version=*device*:1.1. The notation "m.X" used as a revision selects all revisions with the major revision number m. Thus --silicon_version=5510:2.X indicates revisions 2.0, 2.1, etc. of the 5510 device.

This information allows the tools to enable features available to specific devices or revisions if all specified targets have the feature. Similarly, any hardware defect workarounds or detections will only be done if they apply to the designated target(s).

You can safely generate code for your device without specifying the exact revision. For instance, you can use --silicon_version=5509 to generate optimized code compatible with all versions of the C5509. If you want to further optimize to the exact revision in your product, you must know and specify that particular revision number using the --silicon_version option.

If your software is to run on a product or line of products that use different device revisions or even different devices for different models of the product, you can specify multiple device revisions (for example, --silicon_version=5510:2.1 --silicon_version=5510:2.2 --silicon_version=5509) to produce code suitable for the designated hardware.

Generally, using --silicon_version to restrict code generation to a particular set of devices results in better performing code.

The devices and revisions currently supported can be displayed by using the option --silicon_version=list.

The special device name "cpu" can be used to specify a particular revision of the C55x core CPU. If a device of interest does not appear in the output of --silicon_version=list, then you can use --silicon_version=cpu:n after determining n, the CPU revision used in the device.

Compiling For P2 Reserved Mode Using the -vP2 Option

NOTE: When compiling for P2 reserved mode using the -vP2 option, algebraic assembly code is used for both compiler output and inline assembly. The generated code includes workarounds for silicon exceptions in both P2 and P2+ reserved modes. When compiling for P2 reserved mode using -vP2+, only P2+ silicon exception workarounds are included. A separate run-time library, rts552.lib, is provided for P2 and P2+ reserved mode silicon devices.

2.3.5 Symbolic Debugging and Profiling Options

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

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

--profile:power Enables power profiling support by inserting NOPs into the frame code. These NOPs can then be instrumented by the power profiling tooling to track the power usage of functions. If the power profiling tool is not used, this option increases the cycle count of each function because of the NOPs. The --profile:power option also disables optimizations that cannot be handled by the power-profiler.

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

--symdebug:dwarf Generates directives that are used by the C/C++ source-level debugger and enables assembly source debugging in the assembler. The --symdebug:dwarf option's short form is -g. The --symdebug:dwarf option disables many code generator optimizations, because they disrupt the debugger. You can use the --symdebug:dwarf option with the --opt_level (aliased as -O) option to maximize the amount of optimization that is compatible with debugging (see Section 3.8.1).

For more information on the DWARF debug format, see *The DWARF Debugging Standard*.

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

--symdebug:profile_coff Adds the necessary debug directives to the object file which are needed by the profiler to allow function level profiling with minimal impact on optimization (when used). Using --symdebug:coff may hinder some optimizations to ensure that debug ability is maintained, while this option will not hinder optimization.

You can set breakpoints and profile on function-level boundaries in Code Composer Studio, but you cannot single-step through code as with full debug ability.

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

See Section 2.3.12 for a list of deprecated symbolic debugging options.

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 .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:

```
cl55 *.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:

```
cl55 --c_file=file.s --asm_file=assy
```

You cannot use the filename options with wildcard specifications.

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:

```
cl55 --asm_extension=.rrr --obj_extension=.o fit.rrr
```

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

```
cl55 --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:

```
cl55 --abs_directory=d:\abso_list
```

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

```
cl55 --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:

```
cl55 --list_directory=d:\listing
```

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

```
cl55 --obj_directory=d:\object
```

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

```
cl55 --output_file=transfer
```

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

```
cl55 --pp_directory=d:\preproc
```

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

```
cl55 --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 *TMS320C55x Assembly Language Tools User's Guide.*

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

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

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

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

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

--asm_listing Produces an assembly listing file.

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

--assert_arms_set Asserts that assembly code should be compiled as if the ARMS status bit will be set during execution.

--assert_c54cm_set (C54x compatibility mode) Informs the assembler that the C54CM status bit is enabled during the execution of this source file. By default, the assembler assumes that the bit is disabled.

--assert_cpl_set (CPL mode) Informs the assembler that the CPL status bit is enabled during the execution of this source file. This causes the assembler to enforce the use of SP-relative addressing syntax. By default, the assembler assumes that the bit is disabled.

--assert_sst_zero Informs the assembler that the SST status bit is disabled during the execution of this ported C54x source file. By default, the assembler assumes that the bit is enabled.

--bus_conflict (ARMS mode) Informs the assembler that the ARMS status bit is enabled during the execution of this source file. By default, the assembler assumes that the bit is disabled.

--copy_file=*filename* Copies the specified file for the assembly module; acts like a .copy directive. The file is inserted before source file statements. The copied file appears in the assembly listing files.

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

--hash_optional_shift_count (Mnemonic assembly only). Loosens the requirement that a literal shift count operand begin with a # character. This provides compatibility with early versions of the mnemonic assembler. When this option is used and the # is omitted, a warning is issued advising you to change to the new syntax.

--include_file=*filename* Includes the specified file for the assembly module; acts like a .include directive. The file is included before source file statements. The included file does not appear in the assembly listing files.

--no_mac_expand Prevents the expansion of assembly macros when source is output.

--no_nops_in_delay Causes the assembler to remove NOPs located in the delay slots of C54x delayed branch/call instructions.

--output_all_syms Puts labels in the symbol table. Label definitions are written to the COFF symbol table for use with symbolic debugging.

--port_for_speed Causes the assembler to generate faster code rather than smaller code when porting your C54x files. By default, the assembler tries to encode for small code size.

--purecirc Informs the assembler that the C54x file uses C54x circular addressing (does not use the C55x linear/circular mode bits)

--suppress_asm_warnings (Algebraic assembly only). Suppresses assembler warning messages.

--suppress_remark=*num* Suppresses the assembler remark identified by *num*. A remark is an informational assembler message that is less severe than a warning. If you do not specify a value for num, all remarks will be suppressed.

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

--use_long_branch Causes the assembler to use the largest (P24) form of certain variable-length instructions. By default, the assembler tries to resolve all variable-length instructions to their smallest size.

2.3.12 Deprecated Options

Several compiler options have been deprecated. The compiler continues to accept these options, but they are not recommended for use. Future releases of the tools will not support these options. Table 2-29 lists the deprecated options and the options that have replaced them.

Table 2-29. Compiler Backwards-Compatibility Options Summary

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

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

The -mb option is also deprecated for C55x. The option is accepted without an error, but it has no effect on compilation. This option existed to facilitate generation of the dual MAC, MAS, and MPY instructions which on C55x required the memory operands to be in onchip memory.

2.4 Controlling the Compiler Through Environment Variables

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

NOTE: C_OPTION and C_DIR

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

2.4.1 Setting Default Compiler Options (C55X_C_OPTION)

You might find it useful to set the compiler, assembler, and linker default options using the C55X_C_OPTION environment variable. If you do this, the compiler uses the default options and/or input filenames that you name C55X_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 C55X_C_OPTION environment variable and processes it.

The table below shows how to set the C55X_C_OPTION environment variable. Select the command for your operating system:

Operating System	Enter
UNIX (Bourne shell)	**C55X_C_OPTION="** *option*$_1$ [*option*$_2$ . . .]**"; export C55X_C_OPTION**
Windows	**set C55X_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 C55X_C_OPTION environment variable as follows:

```
set C55X_C_OPTION=--quiet --src_interlist --run_linker
```

In the following examples, each time you run the compiler, it runs the linker. Any options following --run_linker on the command line or in C55X_C_OPTION are passed to the linker. Thus, you can use the C55X_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 C55X_C_OPTION is set as shown above:

```
cl55 *c                      ; compiles and links
cl55 --compile_only *.c      ; only compiles
cl55 *.c --run_linker lnk.cmd   ; compiles and links using a command file
cl55 --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 *TMS320C55x Assembly Language Tools User's Guide*.

2.4.2 Naming an Alternate Directory (C55X_C_DIR)

The linker uses the C55X_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)	**C55X_C_DIR="** *pathname$_1$* **;** *pathname$_2$* **;..."; export C55X_C_DIR**
Windows	**set C55X_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 C55X_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 C55X_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 C55X_C_DIR`
Windows	`set C55X_C_DIR=`

2.5 Controlling the Preprocessor

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

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

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

2.5.1 Predefined Macro Names

The compiler maintains and recognizes the predefined macro names listed in Table 2-30.

Table 2-30. Predefined C55x 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
_ _HUGE_MODEL_ _	Defined if huge-model code is selected (the --memory_model=huge option is used); otherwise, it is undefined.
_ _LARGE_MODEL_ _	Defined if large-model code is selected (the --memory_model=large option is used); otherwise, it is undefined.
_ _LINE_ _[1]	Expands to the current line number
_ _SMALL_MODEL_ _	Defined if small-model code is selected (the --memory_model=small option is used); otherwise, it is undefined.

[1] Specified by the ISO standard

Table 2-30. Predefined C55x Macro Names (continued)

Macro Name	Description
_ _STDC_ _[1]	Defined to indicate that compiler conforms to ISO C Standard. See Section 5.1 for exceptions to ISO C conformance.
_ _STDC_VERSION_ _	C standard macro
_ _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_GNU_ATTRIBUTE_SUPPORT_ _	Defined if GCC extensions are enabled (the --gcc option is used); otherwise, it is undefined.
_ _TI_STRICT_ANSI_MODE__	Defined if strict ANSI/ISO mode is enabled (the --strict_ansi option is used); otherwise, it is undefined.
_ _TIME_ _[1]	Expands to the compilation time in the form "*hh:mm:ss*"
_ _TI_RUNTIME_CPP_ _	Defined if --runtime=cpp linker option is used; undefined otherwise.
_ _TI_RUNTIME_DNK_ _	Defined if --runtime=dnk linker option is used; undefined otherwise.
_ _TI_RUNTIME_RTS_ _	Defined if --runtime=rts linker option is used; undefined otherwise.
_ _TMS320C55X_ _	Always defined
_INLINE	Expands to 1 if optimization is used (--opt_level or -O option); undefined otherwise. Regardless of any optimization, always undefined when --no_inlining is used.

You can use the names listed in Table 2-30 in the same manner as any other defined name. For example,

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

translates to a line such as:

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

2.5.2 The Search Path for #include Files

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

- If you enclose the filename in double quotes (" "), the compiler searches for the file in the following directories in this order:
  1. The directory 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 C55X_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 C55X_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 Changing the #include File Search Path (--include_path Option)

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

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

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

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

Assume that the complete pathname for alt.h is:

UNIX	/tools/files/alt.h
Windows	c:\tools\files\alt.h

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

Operating System	Enter
UNIX	`cl55 --include_path=/tools/files source.c`
Windows	`cl55 --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 C55X_C_DIR environment variable.

For example, if you set up C55X_C_DIR with the following command:

```
C55X_C_DIR "/usr/include;/usr/ucb"; export C55X_C_DIR
```

or invoke the compiler with the following command:

```
cl55 --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 Generating a Preprocessed Listing File (--preproc_only Option)

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

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

2.5.4 Continuing Compilation After Preprocessing (--preproc_with_compile Option)

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

2.5.5 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.

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

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

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

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

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

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

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

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

2.6 Understanding Diagnostic Messages

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

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

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

Diagnostic messages have an associated severity, as follows:

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

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

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

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

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

Long messages are wrapped to additional lines, when necessary.

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

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

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

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

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

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

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

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

2.6.1 Controlling Diagnostics

The C/C++ compiler provides diagnostic options to control compiler- and linker-generated diagnostics. The diagnostic options must be specified before the --run_linker option.

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

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

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

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

--display_error_number Displays a diagnostic's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (--diag_suppress, --diag_error, --diag_remark, and --diag_warning). This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix -D; otherwise, no suffix is present. See Section 2.6.

--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 (nonserious warnings), which are suppressed by default.

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

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

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

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

2.6.2 How You Can Use Diagnostic Suppression Options

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

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

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

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

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

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

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

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

2.7 Other Messages

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

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

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

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

sym-id name X filename line number column number

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

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

The --gen_acp_raw option generates a raw listing file that can help you understand how the compiler is preprocessing your source file. Whereas the preprocessed listing file (generated with the --preproc_only, --preproc_with_comment, --preproc_with_line, and --preproc_dependency preprocessor options) shows a preprocessed version of your source file, a raw listing file provides a comparison between the original source line and the preprocessed output. The raw listing file has the same name as the corresponding source file with an *.rl* extension.

The raw listing file contains the following information:

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

Each source line in the raw listing file begins with one of the identifiers listed in Table 2-31.

Table 2-31. Raw Listing File Identifiers

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

The --gen_acp_raw option also includes diagnostic identifiers as defined in Table 2-32.

Table 2-32. 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 2-32 that indicates the severity of the diagnostic
filename	The source file
line number	The line number in the source file
column number	The column number in the source file
diagnostic	The message text for the diagnostic

Diagnostics after the end of file are indicated as the last line of the file with a column number of 0. When diagnostic message text requires more than one line, each subsequent line contains the same file, line, and column information but uses a lowercase version of the diagnostic identifier. For more information about diagnostic messages, see Section 2.6.

2.10 Using Inline Function Expansion

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

There are several types of inline function expansion:

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

NOTE: Function Inlining Can Greatly Increase Code Size

Expanding functions inline increases code size, especially inlining a function that is called in a number of places. Function inlining is optimal for functions that are called only from a small number of places and for small functions.

2.10.1 Inlining Intrinsic Operators

There are many intrinsic operators for the C55x. The compiler replaces intrinsic operators with efficient code (usually one instruction). All of the operators are automatically inlined by the compiler. The inlining happens automatically whether or not you use the optimizer.

For details about intrinsics, and a list of the intrinsics, see Section 6.5.5.

Additional functions that may be expanded inline are:

- abs
- labs
- fabs
- assert
- _nassert
- memcpy

2.10.2 Unguarded Definition-Controlled Inlining

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

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

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

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

Example 2-1. Using the Inline Keyword

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

2.10.3 Guarded Inlining and the _INLINE Preprocessor Symbol

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

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

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

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

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

Example 2-2. Header File string.h

```
/*****************************************************************************/
/* string.h   v x.xx                                                         */
/* Copyright (c) 2002 Texas Instruments Incorporated                         */
/*****************************************************************************/

; . . .
#ifndef   _SIZE_T
#define   _SIZE_T
typedef   unsigned int size_t;
#endif

#ifdef    _INLINE
#define   __INLINE static inline
#else
#define   __INLINE
#endif

__INLINE size_t  strlen(const char *_string);
; . . .

#ifdef    _INLINE

/*****************************************************************************/
/*  strlen                                                                   */
```

Example 2-2. Header File string.h (continued)

```
/*****************************************************************************/
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
#undef    __INLINE
#endif
```

Example 2-3. Library Definition File

```
/*****************************************************************************/
/*  strlen v x.xx                                                            */
/*  Copyright (c) 2002  Texas Instruments Incorporated                       */
/*****************************************************************************/
#undef _INLINE
#include <string.h>

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

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


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

2.10.4 Inlining Restrictions

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

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

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

2.11 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:

```
cl55 --c_src_interlist function
```

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

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

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

Example 2-4 shows a typical interlisted assembly file.

For more information about using the interlist feature with the optimizer, see Section 3.7.

Example 2-4. An Interlisted Assembly Language File

```
     .global  _main
;-------------------------------------------------------------------------
;     3 | void main (void)
;-------------------------------------------------------------------------
;*************************************************************************
;* FUNCTION NAME       :  _main                                          *
;* Stack Frame         :  Compact (No Frame Pointer, w/ debug)           *
;* Total Frame Size    :  3 words                                        *
;*                        (1 return address/alignment)                   *
;*                        (2 function parameters)                        *
;*************************************************************************
_main:
        AADD #-3, SP
;------------------------------------------------------------------------
;     5 | printf("Hello World\n");
;------------------------------------------------------------------------
          MOV #SL1, *SP(#0)
          CALL      #_printf  ;
                              ; call occurs [#_printf];
          AADD #3, SP
          RET                 ; return occurs
;************************************************************************
;* STRINGS                                                              *
;************************************************************************
          .sect  ".const"
          .align 1
          SL1:   .string  "Hello World",10,0
;************************************************************************
;* UNDEFINED EXTERNAL REFERENCES                                        *
;************************************************************************

          .global  _printf
```

2.12 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 5.9.21 for information about the NO_HOOKS pragma.

Chapter 3

SPRU281G–December 2011

Optimizing Your Code

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

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

Topic **Page**

3.1 Invoking Optimization

The C/C++ compiler is able to perform various optimizations. High-level optimizations are performed in the optimizer and low-level, target-specific optimizations occur in the code generator. Use high-level 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, and 3), which controls the type and degree of optimization.

- **--opt_level=0** or **-O0**
  - Performs control-flow-graph simplification
  - Allocates variables to registers
  - Performs loop rotation
  - Eliminates unused code
  - Simplifies expressions and statements
  - Expands calls to functions declared inline
- **--opt_level=1** or **-O1**

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

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

  The optimizer uses --opt_level=2 (-O2) as the default if you use --opt_level (-O) without an optimization level.
- **--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 3.2 and Section 3.3 for more information.

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

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

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

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

If You ...	Use this Option	See
Have files that redeclare standard library functions	--std_lib_func_defined --std_lib_func_redefined	Section 3.2.1
Want to create an optimization information file	--gen_opt_level=*n*	Section 3.2.2
Want to compile multiple source files	--program_level_compile	Section 3.3

3.2.1 Controlling File-Level Optimization (--std_lib_func_def Options)

When you invoke the compiler with the --opt_level=3 option, some of the optimizations use known properties of the standard library functions. If your file redeclares any of these standard library functions, these optimizations become ineffective. Use Table 3-2 to select the appropriate file-level optimization option.

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

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

3.2.2 Creating an Optimization Information File (--gen_opt_info Option)

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

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

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

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

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

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

To see which program-level optimizations the compiler is applying, use the --gen_opt_level=2 option to generate an information file. See Section 3.2.2 for more information.

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

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

NOTE: 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.

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

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

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

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

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

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

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

In some situations when you use --program_level_compile and --opt_level=3, you *must* use a --call_assumptions option or the FUNC_EXT_CALLED pragma. See Section 3.3.2 for information about these situations.

3.3.2 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 5.9.11) 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 3.3.1).

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

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

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

Solution — Compile with --program_level_compile --opt_level=3 --call_assumptions=2 to tell the compiler that outside functions do not call C/C++ functions or modify C/C++ variables. See Section 3.3.1 for information about the --call_assumptions=2 option.

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

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

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

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

See Section 3.3.1 for information about the --call_assumptions=*n* option.

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

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

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

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

3.4 Indicating Whether Certain Aliasing Techniques Are Used

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

The following sections describe some aliasing techniques that may be used in your code. These techniques are valid according to the ISO C standard and are accepted by the C55 compiler; however, they prevent the optimizer from fully optimizing your code.

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

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

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

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

```
int *glob_ptr;

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

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

    h(x);
}

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

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

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

- The --no_bad_aliases option indicates that your code does not use the aliasing technique described in Section 3.4.1. If your code uses that technique, do *not* use the --no_bad_aliases option. You must compile with the --aliased_variables option.

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

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

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

    p[2] = 5;
}
```

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

```
g(int j)
{

    int a[20];

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

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

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

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

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

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

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

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

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

3.5 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.

3.6 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.

Optimization Level 3 and Inlining

NOTE: 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.

Inlining and Code Size

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

3.7 Using the Interlist Feature With Optimization

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

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

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

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

Example 3-1 shows a function that has been compiled with optimization (--opt_level=2) and the --optimizer_interlist option. The assembly file contains compiler comments interlisted with assembly code.

Impact on Performance and Code Size

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

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

Example 3-2 shows the function from Example 3-1 compiled with the optimization (--opt_level=2) and the --c_src_interlist and --optimizer_interlist options. The assembly file contains compiler comments and C source interlisted with assembly code.

Example 3-1. The Function From Example 2-4 Compiled With the -O2 and --optimizer_interlist Options

```
_main:
;** 5  ------------------ printf((char *)"Hello, world\n");
;** 5  ------------------ return;
       AADD #-3, SP
       MOV #SL1, *SP(#0)
       CALL #_printf
                          ; call occurs [#_printf]
       AADD #3, SP
       RET                ;return occurs
```

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

```
_main:
;** 5  -----------------  printf((char *)"Hello, world\n");
;**    -----------------  return;
          AADD #-3, SP
;-------------------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;-------------------------------------------------------------------------------
          MOV #SL1, *SP(#0)
          CALL #_printf
                          ; call occurs [#_printf]
          AADD #3, SP
          RET             ;return occurs
```

3.8 Debugging and Profiling Optimized Code

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

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

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

If you want to use symbolic debugging and still generate fully optimized code, use the --optimize_with_debug option. This option reenables the optimizations disabled by --symdebug:dwarf or --symdebug:coff. However, if you use the --optimize_with_debug option, portions of the debugger's functionality will be unreliable.

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

Symbolic Debugging Options Affect Performance and Code Size

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

3.8.2 Profiling Optimized Code

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

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

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

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

Profile Points

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

3.9 Controlling Code Size Versus Speed

The latest mechanism for controlling the goal of optimizations in the compiler is represented by the --opt_for_speed=*num* option. The *num* denotes the level of optimization (0-5), which controls the type and degree of code size or code speed optimization:

- --opt_for_speed=0

  Enables optimizations geared towards improving the code size with a *high* risk of worsening or impacting performance.
- --opt_for_speed=1

  Enables optimizations geared towards improving the code size with a *medium* risk of worsening or impacting performance.
- --opt_for_speed=2

  Enables optimizations geared towards improving the code size with a *low* risk of worsening or impacting performance.
- --opt_for_speed=3

  Enables optimizations geared towards improving the code performance/speed with a *low* risk of worsening or impacting code size.
- --opt_for_speed=4

  Enables optimizations geared towards improving the code performance/speed with a *medium* risk of worsening or impacting code size.
- --opt_for_speed=5

  Enables optimizations geared towards improving the code performance/speed with a *high* risk of worsening or impacting code size.

If you specify the option without a parameter, the default setting is --opt_for_speed=4. However, the default behavior of the compiler is as if --opt_for_speed=1 were specified.

The initial 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

3.10 What Kind of Optimization Is Being Performed?

The TMS320C55x 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 3.10.1
Alias disambiguation	Section 3.10.1
Branch optimizations and control-flow simplification	Section 3.10.3
Data flow optimizations • Copy propagation • Common subexpression elimination • Redundant assignment elimination	Section 3.10.4
Expression simplification	Section 3.10.5
Inline expansion of functions	Section 3.10.6
Function Symbol Aliasing	Section 3.10.7
Induction variable optimizations and strength reduction	Section 3.10.8
Loop-invariant code motion	Section 3.10.9
Loop rotation	Section 3.10.10
Instruction scheduling	Section 3.10.11

C55x-Specific Optimization	See
Tail merging	Section 3.10.12
Autoincrement addressing	Section 3.10.13
Hardware loop instructions	Section 3.10.14
Circular addressing	Section 3.10.15

3.10.1 Cost-Based Register Allocation

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

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

3.10.2 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.

3.10.3 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.

3.10.4 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.

3.10.5 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.

3.10.6 Inline Expansion of Functions

The compiler replaces calls to small functions with inline code, saving the overhead associated with a function call as well as providing increased opportunities to apply other optimizations.

3.10.7 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.

3.10.8 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.

3.10.9 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.

3.10.10 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.

3.10.11 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 pipeline latencies. It can also be used to reduce code size.

3.10.12 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.

3.10.13 Autoincrement Addressing

For pointer expressions of the form *p++, the compiler uses efficient C55x 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)...
```

3.10.14 Hardware Loop Instructions

The C55x supports zero-overhead loops with the RPTB (repeat block), RPTBLOCAL and RPT (repeat single) instructions. With the optimizer, the compiler can detect loops controlled by counters and generate them using the efficient repeat forms. The iteration count can be either a constant or an expression.

Induction variable elimination and loop test replacement allow the compiler to recognize the loop as a simple counting loop and then generate a hardware loop instruction. Strength reduction turns the array references into efficient pointer autoincrements.expression. Induction

3.10.15 Circular Addressing

The compiler can generate assembly code that utilizes the circular addressing hardware of the C55x. The compiler can transform certain array references inside loops into circular addressing code. Given an array *a* of size *S* and index variable *x*, the compiler will recognize *a* as a circular buffer if the following conditions hold:

- All index expressions for *a* contain only *x* and constants. An index expressions must be of the form *bx* + *c* where *b* and *c* are constants.
- Increments of *x* are always by positive constants.
- Increments of *x* are always followed by `% S` (i.e. modulus the size of the buffer). The syntax & *T* is also acceptable where *T* = *S* – 1 and *S* is a power of two.
- *x* is initialized with a value that is a compile-time constant.

Example 3-3 shows a simple C function for which the compiler can make use of the C55x circular addressing hardware. The compiler detects a circular buffer with array b, index variable x and size 16.

Example 3-3. A Simple Circular Addressing Example

```
void circ(int *a, int *b)
{
    int i, x = 0;

    for(i = 0; i < 16; i++) /*(1) start of circular buffer lifetime */
    {
        a[i] = b[x];
        x = (x + 3) % 16;    /* or x = (x + 3) & 15 */
    }                        /* (2) end of circular buffer lifetime */
}
```

Bear in mind the following restrictions on circular addressing:

- A negative update of x (e.g. x = (x – 1) % 10) is not allowed and would prohibit the compiler from generating circular addressing code. However, a negative update can be simulated by way of a large positive update. Subtracting one from a circular buffer of size ten would look like this x = (x + 9) % 10.
- The circular buffer size must be a compile time constant. That is, the compiler must be able to determine what the size of the buffer is, or it does not generate circular addressing code.
- The compiler does not support circular addressing via the CDP register.
- The hardware has a limited number of registers that it can use for circular addressing. If you write code that requires more than the available number of circular registers, the compiler attempts to generate efficient code to simulate circular addressing for some circular buffers.
- The compiler does not generate circular addressing code if a function call would be present within the lifetime of the circular buffer (that is, no function calls are allowed between (1) and (2) in Example 3-3.)

Some of these restrictions may be eased in future revisions of the compiler.

The circular addressing hardware can also be accessed using the _circ_incr intrinsic (see Section 6.5.5.1)

Chapter 4

SPRU281G–December 2011

Linking C/C++ Code

The C/C++ compiler and assembly language tools provide two methods for linking your programs:

- You can compile individual modules and link them together. This method is especially useful when you have multiple source files.
- You can compile and link in one step. This method is useful when you have a single source module.

This chapter describes how to invoke the linker with each method. It also discusses special requirements of linking C/C++ code, including the run-time-support libraries, specifying the type of initialization, and allocating the program into memory. For a complete description of the linker, see the *TMS320C55x 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:

cl55 --run_linker {**--rom_model | --ram_model**} *filenames*
[*options*] [**--output_file=** *name.out*] **--library=** *library* [*lnk.cmd*]

cl55 --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 cl55 --run_linker, you must use **--rom_model** or **--ram_model**. The --rom_model option uses automatic variable initialization at run time; the --ram_model option uses variable initialization at load time.

filenames Names of object files, linker command files, or archive libraries. The default extension for all input files is *.obj*; any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is *a.out*, unless you use the --output_file option to name the output file.

options Options affect how the linker handles your object files. Linker options can only appear after the **--run_linker** option on the command line, but otherwise may be in any order. (Options are discussed in detail in the *TMS320C55x 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.

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 *TMS320C55x Assembly Language Tools User's Guide*.

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

```
cl55 --run_linker --rom_model prog1 prog2 prog3 --output_file=prog.out
     --library=rts55.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:

cl55*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:

```
cl55 prog1.c prog2.c prog3.c --run_linker --rom_model --output_file=prog.out --library=rts55.lib
```

NOTE: Order of Processing Arguments in the Linker

The order in which the linker processes arguments is important. The compiler passes arguments to the linker in the following order:

1. Object filenames from the command line
2. Arguments following the --run_linker option on the command line
3. Arguments following the --run_linker option from the C55X_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 C55X_C_OPTION environment variable and want to selectively disable linking with the --compile_only option on the command line.

4.2 Linker Code Optimizations

These options are used to further optimize your code.

4.2.1 Generate List of Dead Functions (--generate_dead_funcs_list Option)

In order to facilitate the removal of unused code, the linker generates a feedback file containing a list of functions that are never referenced. The feedback file must be used the next time you compile the source files. The syntax for the --generate_dead_funcs_list option is:

--generate_dead_funcs_list= *filename*

If *filename* is not specified, a default filename of dead_funcs.txt is used.

Proper creation and use of the feedback file entails the following steps:

1. Compile all source files using the --gen_func_subsections compiler option. For example:

   ```
   cl55 file1.c file2.c --gen_func_subsections
   ```

2. During the linker, use the --generate_dead_funcs_list option to generate the feedback file based on the generated object files. For example:

   ```
   cl55 --run_linker file1.obj file2.obj --generate_dead_funcs_list=feedback.txt
   ```

   Alternatively, you can combine steps 1 and 2 into one step. When you do this, you are not required to specify --gen_func_subsections when compiling the source files as this is done for you automatically. For example:

   ```
   cl55 file1.c file2.c --run_linker --generate_dead_funcs_list=feedback.txt
   ```

3. Once you have the feedback file, rebuild the source. Give the feedback file to the compiler using the --use_dead_funcs_list option. This option forces each dead function listed in the file into its own subsection. For example:

   ```
   cl55 file1.c file2.c --use_dead_funcs_list=feedback.txt
   ```

4. Invoke the linker with the newly built object files. The linker removes the subsections. For example:

   ```
   cl55 --run_linker file1.obj file2.obj
   ```

   Alternatively, you can combine steps 3 and 4 into one step. For example:

   ```
   cl55 file1.c file2.c --use_dead_funcs_list=feedback.txt --run_linker
   ```

NOTE: Dead Functions Feedback

The format of the feedback file generated with --gen_dead_funcs_list is tightly controlled. It must be generated by the linker in order to be processed correctly by the compiler. The format of this file may change over time, so the file contains a version format number to allow backward compatibility.

4.2.2 Generating Function Subsections (--gen_func_subsections Compiler Option)

When the linker places code into an executable file, it allocates all the functions in a single source file as a group. This means that if any function in a file needs to be linked into an executable, then all the functions in the file are linked in. This can be undesirable if a file contains many functions and only a few are required for an executable.

This situation may exist in libraries where a single file contains multiple functions, but the application only needs a subset of those functions. An example is a library .obj file that contains a signed divide routine and an unsigned divide routine. If the application requires only signed division, then only the signed divide routine is required for linking. By default, both the signed and unsigned routines are linked in since they exist in the same .obj file.

The --gen_func_subsections compiler option remedies this problem by placing each function in a file in its own subsection. Thus, only the functions that are referenced in the application are linked into the final executable. This can result in an overall code size reduction.

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 *TMS320C55x 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

If the --rom_model or --ram_model option is specified during the linker and the entry point for the program (normally c_int00) is not resolved by any specified object file or library, the linker attempts to automatically include the best compatible run-time-support library for your program. The chosen run-time-support library is linked in after any other libraries specified with the --library option on the command line. Alternatively, you can force the linker to choose an appropriate run-time-support library by specifying “libc.a” as an argument to the --library option, or when specifying the run-time-support library name explicitly in a linker command file.

The automatic selection of a run-time-support library can be disabled with the --disable_auto_rts option.

If the --issue_remarks option is specified before the --run_linker option during the linker, a remark is generated indicating which run-time support library was linked in. If a different run-time-support library is desired, you must specify the name of the desired run-time-support library using the --library option and in your linker command files when necessary.

Example 4-1. Using the --issue_remarks Option

```
cl55 --issue_remarks main.c --run_linker --rom_model

<Linking>

remark: linking in "libc.a"

remark: linking in "rts55.lib" in place of "libc.a"
```

4.3.1.2 Manual Run-Time-Support Library Selection

You should use the --library linker option to specify which C55x run-time-support library to use. The --library option also tells the linker to look at the --search_path options and then the C55X_C_DIR environment variable to find an archive path or object file. To use the --library linker option, type on the command line:

cl55 --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 and secondary system 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)
5. Call the function main
6. Call exit when main returns

A sample bootstrap routine is _c_int00, provided in boot.obj in rts55.lib. The entry point is usually set to the starting address of the bootstrap routine.

NOTE: The _c_int00 Symbol

If you use the --ram_model or --rom_model link option, _c_int00 is automatically defined as the entry point for the program.

4.3.3 Initialization by the Interrupt Vector

If your program is expected to run from RESET, you must set up the reset vector to branch to _c_int00. Referring to _c_int00 causes boot.obj to be loaded from the library. The _c_int00 routine will initialize the program's state. The boot.obj code places the address of _c_int00 into a section named .reset. The section can then be allocated at the reset vector location using the linker.

A sample interrupt vector is provided in vectors.obj in rts55.lib. For C55x, 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 function main is called. Global destructors are invoked during the function exit, similar to functions registered through atexit.

Section 6.9.1.4 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 6.9.1.1 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 6.9.1.2).
- Global variables are initialized at *load time*. Use the --ram_model linker option (see Section 6.9.1.3).

When you link a C/C++ program, you must use either the --rom_model or --ram_model option. These options tell the linker to select initialization at run time or load time.

When you compile and link programs, the --rom_model option is the default. If used, the --rom_model option must follow the --run_linker option (see Section 4.1). The following list outlines the linking conventions used with --rom_model or --ram_model:

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

NOTE: Boot Loader

A loader is not included as part of the C/C++ compiler tools. You can use the C28x simulator or emulator with the source debugger as a loader.

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.

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 Created by the Compiler

Name	Contents
.cinit	Tables for explicitly initialized global and static variables
.const	Global and static const variables that are explicitly initialized and contain string literals. String literals, other than those used in an array initializer, are placed in the .const:.string subsection to enable greater link-time placement control.
.pinit	Table of constructors to be called at startup
.text	Executable code and constants

Table 4-2. Uninitialized Sections Created by the Compiler

Name	Contents
.args	Linker-created section used to pass arguments from the command line of the loader to the program
.bss	Global and static variables
.stack	Primary stack
.sysmem	Memory for malloc functions (heap)
.sysstack	Secondary system stack

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. See Section 6.1.4 for a complete description of how the compiler uses these sections.

The linker provides MEMORY and SECTIONS directives for allocating sections. For more information about allocating sections into memory, see the *TMS320C55x Assembly Language Tools User's Guide*.

Allocating Sections

NOTE: When allocating sections, keep in mind that the .stack and .sysstack sections must be on the same 64K-word data page. Also, only code sections and huge-model data sections are allowed to cross page boundaries.

4.3.7 A Sample Linker Command File

Example 4-2 shows a typical linker command file that links a C/C++ program. The command file in this example is named lnk.cmd and lists several linker options.

To link the program, use the following syntax:

cl55 --run_linker *object_file(s)* **--output_file=** *outfile* **--map_file=** *mapfile* **lnk.cmd**

The MEMORY and possibly the SECTIONS directive might require modification to work with your system. See the *Linker Description* chapter of the *TMS320C55x Assembly Language Tools User's Guide* for information on these directives.

Example 4-2. Linker Command File

```
--stack_size  0x2000 /* PRIMARY STACK SIZE                                    */
--sysstack    0x1000 /* SECONDARY STACK SIZE                                  */
--heap_size   0x2000 /* HEAP AREA SIZE                                        */
--rom_model          /* Use C linking conventions: auto-init vars at run time */
-u_Reset             /* Force load of reset interrupt handler                 */


/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
 PAGE 0:  /* ---- Unified Program/Data Address Space ---- */
   RAM (RWIX): origin = 0x000100, length = 0x01FF00    /* 128Kb page of RAM   */
   ROM (RIX) : origin = 0x020100, length = 0x01FF00    /* 128Kb page of ROM   */
   VECS (RIX): origin = 0xFFFF00, length = 0x000100    /* 256-byte int vector */

 PAGE 2: /* ------ 64K-word I/O Address Space ------- */
   IOPORT (RWI) : origin = 0x000000, length = 0x020000
}
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */

SECTIONS
{
   .text      > ROM  PAGE 0    * CODE                                */
 /* These sections must be on same physical memory page when        */
 /* small memory model is used                                       */
   .data      > RAM  PAGE 0    /* INITIALIZED VARS                   */
   .bss          > RAM  PAGE 0    /* GLOBAL & STATIC VARS               */
   .const     > RAM  PAGE 0    /* CONSTANT DATA                      */
   .sysmem    > RAM  PAGE 0    /* DYNAMIC MEMORY (malloc)            */
   .stack     > RAM  PAGE 0    /* PRIMARY SYSTEM STACK               */
   .sysstack  > RAM  PAGE 0    /* SECONDARY SYSTEM STACK             */
   .cio       > RAM  PAGE 0    /* C I/O BUFFERS                      */
 /* The .switch, .cinit, and .pinit sections can be on any physical  */
 /* memory page when small memory model is used                      */
   .switch    > RAM  PAGE 0    /* SWITCH STATEMENT TABLES            */
   .cinit     > RAM  PAGE 0    /* AUTOINITIALIZATION TABLES          */
   .pinit     > RAM  PAGE 0    /* INITIALIZATION FN TABLES           */

   vectors    > VECS PAGE 0    /* INTERRUPT VECTORS                  */

   .ioport    > IOPORT PAGE 2  /* GLOBAL & STATIC IO VARS            */
}
```

Chapter 5

SPRU281G–December 2011

TMS320C55x C/C++ Language Implementation

The C/C++ compiler supports the C/C++ language standard that was developed by a committee of the American National Standards Institute (ANSI) and subsequently adopted by the International Standards Organization (IS0).

The C++ language supported by the C55x is defined by the ANSI/ISO/IEC 14882:1998 standard with certain exceptions.

Topic **Page**

5.1 Characteristics of TMS320C55x C

The compiler supports the C language as defined by ISO/IEC 9899:1990, which is equivalent to American National Standard for Information Systems-Programming Language C X3.159-1989 standard, commonly referred to as C89, published by the American National Standards Institute. The compiler can also accept many of the language extensions found in the GNU C compiler (see Section 5.14). The compiler does not support C99.

The ANSI/ISO standard identifies some features of the C language that are affected by characteristics of the target processor, run-time environment, or host environment. For reasons of efficiency or practicality, this set of features can differ among standard compilers.

Unsupported features of the C library are:

- The run-time library has minimal support for wide and multi-byte characters. The type wchar_t is implemented as int. The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h>, but does not include all the functions specified in the standard. So-called multi-byte characters are limited to single characters. There are no shift states. The mapping between multi-byte characters and wide characters is simple equivalence; that is, each wide character maps to and from exactly a single multi-byte character having the same value.
- The run-time library includes the header file <locale.h>, but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hard-coded to the behavior of the C locale, and attempting to install a different locale by way of a call to setlocale() will return NULL.

5.2 Characteristics of TMS320C55x C++

The C55x compiler supports C++ as defined in the ANSI/ISO/IEC 14882:1998 standard, including these features:

- Complete C++ standard library support, with exceptions noted below.
- Templates
- Exceptions, which are enabled with the --exceptions option; see Section 5.6.
- Run-time type information (RTTI), which can be enabled with the --rtti compiler option.

The *exceptions* to the standard are as follows:

- 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.
- No support for bad_cast or bad_type_id is included in the typeinfo header.
- Two-phase name binding in templates, as described in [tesp.res] and [temp.dep] of the standard, is not implemented.
- The export keyword for templates is not implemented.
- A typedef of a function type cannot include member function cv-qualifiers.
- A partial specialization of a class member template cannot be added outside of the class definition.

5.3 Using MISRA-C:2004

You can alter your code to work with the MISRA-C:2004 rules. The following enable/disable the rules:

- The --check_misra option enables checking of the specified MISRA-C:2004 rules.
- The CHECK_MISRA pragma enables/disables MISRA-C:2004 rules at the source level. This pragma is equivalent to using the --check_misra option. See Section 5.9.2.
- RESET_MISRA pragma resets the specified MISRA-C:2004 rules to the state they were before any CHECK_MISRA pragmas were processed. See Section 5.9.22.

The syntax of the option and pragmas is:

--check_misra={all|required|advisory|none|*rulespec*}

#pragma CHECK_MISRA ("{all|required|advisory|none|*rulespec*}**");**

#pragma RESET_MISRA ("{all|required|advisory|*rulespec*}**");**

The *rulespec* parameter is a comma-separated list of these specifiers:

[-]X	Enable (or disable) all rules in topic X.
[-]X-Z	Enable (or disable) all rules in topics X through Z.
[-]X.A	Enable (or disable) rule A in topic X.
[-]X.A-C	Enable (or disable) rules A through C in topic X.

Example: --check_misra=1-5,-1.1,7.2-4

- Checks topics 1 through 5
- Disables rule 1.1 (all other rules from topic 1 remain enabled)
- Checks rules 2 through 4 in topic 7

Two options control the severity of certain MISRA-C:2004 rules:

- The --misra_required option sets the diagnostic severity for required MISRA-C:2004 rules.
- The --misra_advisory option sets the diagnostic severity for advisory MISRA-C:2004 rules.

The syntax for these options is:

--misra_advisory={error|warning|remark|suppress}

--misra_required={error|warning|remark|suppress}

5.4 Data Types

Table 5-1 lists the size, representation, and range of each scalar data type for the C55x compiler. Many of the range values are available as standard macros in the header file limits.h.

Table 5-1. TMS320C55x C/C++ 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
short, signed short	16 bits	2s complement	-32 768	32 767
unsigned short	16 bits	Binary	0	65 535
int, signed int	16 bits	2s complement	-32 768	32 767
unsigned int	16 bits	Binary	0	65 535
long, signed long	32 bits	2s complement	-2 147 483 648	2 147 483 647
unsigned long	32 bits	Binary	0	4 294 967 295
long long, signed long long	40 bits	2s complement	-549 755 813 888	549 755 813 887
unsigned long long	40 bits	Binary	0	1 099 511 627 775
enum (C)	16 bits	2s complement	-32 768	32 767
enum[1] (C++)	16, 32, 40 bits	2s complement	-549 755 813 888	549 755 813 887
float	32 bits	IEEE 32-bit	1.175 494e-38[2]	3.40 282 346e+38
double	32 bits	IEEE 32-bit	1.175 494ee-38[2]	3.40 282 346e+38
long double	32 bits	IEEE 32-bit	1.175 494ee-38[2]	3.40 282 346e+38
pointers (data):			0	
small memory mode	16 bits	Binary	0	0xFFFF
large memory mode	23 bits	Binary	0	0x7FFFFF
pointers (function)	24 bits	Binary	0	0xFFFFFF

[1] The representation is the same as the underlying type (int, long or long long)

[2] Figures are minimum precision.

C55x Byte Is 16 Bits

NOTE: By ISO C definition, the size of operator yields the number of bytes required to store an object. ISO further stipulates that when sizeof is applied to char, the result is 1. Since the C55x char is 16 bits (to make it separately addressable), a byte is also 16 bits. This yields results you may not expect; for example, sizeof (int) == 1 (not 2). C55x bytes and words are equivalent (16 bits).

long long Is 40 Bits

NOTE: The long long data type is implemented according to the ISO/IEC 9899 C Standard. However, the C55x compiler implements this data type as 40 bits instead of 64 bits. Use the “ll” length modifier with formatted I/O functions (such as printf and scanf) to print or read long long variables. For example:

```
printf("%lld\n", (long long)global);
```

5.5 Keywords

The C55x C/C++ compiler supports the standard const, restrict, and volatile keywords. In addition, the C/C++ compiler extends the C/C++ language through the support of the interrupt, ioport, onchip, and restrict keywords.

5.5.1 The const Keyword

The C/C++ compiler supports the ANSI/ISO standard keyword *const*. This keyword gives you greater optimization and control over allocation of storage for certain data objects. You can apply the const qualifier to the definition of any variable or array to ensure that its value is not altered.

If you define an object as const, the .const section allocates storage for the object. The const data storage allocation rule has two exceptions:

- If the keyword volatile is also specified in the definition of an object (for example, volatile const int x). Volatile keywords are assumed to be allocated to RAM. (The program does not modify a const volatile object, but something external to the program might.)
- If the object has automatic storage (function scope).

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

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

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

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

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

5.5.2 The interrupt Keyword

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

Functions that handle interrupts follow special register-saving rules and a special return sequence. 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 function main. Because it has no caller, c_int00 does not save any registers.

Use the alternate keyword, __interrupt, if you are writing code for strict ANSI/ISO mode (using the --strict_ansi compiler option).

HWI Objects and the interrupt Keyword

NOTE: The interrupt keyword must not be used when BIOS HWI objects are used in conjunction with C functions. The HWI_enter/HWI_exit macros and the HWI dispatcher contain this functionality, and the use of the C modifier can cause negative results.

5.5.3 The ioport Keyword

The C55x processor contains a secondary memory space for I/O. The compiler extends the C/C++ language by adding the ioport keyword to support the I/O addressing mode.

The ioport type qualifier can be used with the standard type specifiers including arrays, structures, unions, and enumerations. It can also be used with the const and volatile type qualifiers. When used with an array, ioport qualifies the elements of the array, not the array type itself. Members of structures cannot be qualified with ioport unless they are pointers to ioport data.

The ioport type qualifier can only be applied to global or static variables. Local variables cannot be qualified with ioport. However, a pointer can point to a ioport-qualified type. For example:

```
void foo (void)
{
   ioport int i; /* invalid */
   ioport int *j; /* valid */
```

When declaring a pointer qualified with ioport, note that the meaning of the declaration will be different depending on where the qualifier is placed. Because I/O space is 16-bit addressable, pointers to I/O space are always 16 bits, even in the large memory model.

You cannot use printf() with a direct ioport pointer argument. Instead, pointer arguments in printf() must be converted to “void *”, as shown in the example below:

```
ioport int *p;
printf("%p\n", (void*)p);
```

The declaration shown in Example 5-1 places a pointer in I/O space that points to an object in data memory in Example 5-2.

Example 5-1. Declaring a Pointer in I/O Space

```
int * ioport ioport_pointer; /* ioport pointer */
int i;
int j;

void foo (void)
{
   ioport_pointer = &I;
   j = *ioport_pointer;
}
```

Example 5-2. Compiler Output For Example 5-1

```
_foo:
   MOV #_i,port(#_ioport_pointer)  ; store addr of #i (I/O memory)                              ;
   MOV port(#_ioport_pointer),AR3  ; load address of #i (I/O memory)
   MOV *AR3,AR1                    ; indirectly load value of #i
   MOV AR1,*abs16(#_j)             ; store value of #i at #j
   RET
```

If typedef had been used in Example 5-1, the pointer declaration would have been:

```
typedef int *int_pointer;
ioport int_pointer ioport_pointer; /* ioport pointer */
```

Example 5-3 declares a pointer that points to data in I/O space in Example 5-4. This pointer is 16 bits even in the large memory model.

Example 5-3. Declaring a Pointer That Points to Data in I/O Space

```
/* pointer to ioport data: */
ioport int * ptr_to_ioport;
ioport int i;

void foo (void)
{
   int j;
 i = 10;
   ptr_to_ioport = &I;
   j = *ptr_to_ioport;
}
```

Example 5-4. Compiler Output For Example 5-3

```
_foo:
   MOV #_i,*abs16(#_ptr_to_ioport)  ; store address of #i
   MOV *abs16(#_ptr_to_ioport),AR3
   AADD #-1, SP
   MOV #10,port(#_i)                ; store 10 at #i (I/O memory)
   MOV *AR3,AR1
   MOV AR1,*SP(#0)
   AADD #1,SP
   RET
```

Example 5-5 declares an ioport pointer that points to data in I/O space in Example 5-6.

Example 5-5. Declaring an ioport Pointer That Points to Data in I/O Space

```
/* ioport pointer to ioport data: */
ioport int * ioport iop_ptr_to_ioport;
ioport int i;
ioport int j;

void foo (void)
{
   i = 10;
   iop_ptr_to_ioport = &I;
   j = *iop_ptr_to_ioport;
}
```

Example 5-6. Compiler Output For Example 5-5

```
_foo:
   MOV #10,port(#_i)                 ; store 10 at #i (I/O memory)
   MOV #_i,port(#_iop_ptr_to_ioport) ; store address of
                                     ; #i (I/O memory)
   MOV port(#_iop_ptr_to_ioport),AR3 ; load addr of #i
   MOV *AR3, AR1                     ; load #i
   MOV AR1,port(#_j)                 ; store 10 in #j (I/O memory)
   RET
```

Use the alternate keyword, __ioport, if you are writing code for strict ANSI/ISO mode (using the --strict_ansi compiler option).

5.5.4 The onchip Keyword

The onchip keyword informs the compiler that a pointer points to data that may be used as an operand to a dual MAC instruction. Data passed to a function with onchip parameters, or data that will be eventually referenced through an onchip expression, must be linked into on-chip memory (not external memory). Failure to link the data appropriately can result in a reference to external memory through the BB data bus, which will generate a bus error.

```
onchip int x[100];  /* array declaration   */
onchip int *p;      /* pointer declaration */
```

The -mb compiler option specifies that all data memory will be on-chip.

Use the alternate keyword, __onchip, if you are writing code for strict ANSI/ISO mode (using the --strict_ansi compiler option).

5.5.5 The restrict Keyword

To help the compiler determine memory dependencies, you can qualify a pointer, reference, or array with the restrict keyword. The restrict keyword is a type qualifier that can be applied to pointers, references, and arrays. Its use represents a guarantee by you, the programmer, that within the scope of the pointer declaration the object pointed to can be accessed only by that pointer. Any violation of this guarantee renders the program undefined. This practice helps the compiler optimize certain sections of code because aliasing information can be more easily determined.

In , the restrict keyword is used to tell the compiler that the function func1 is never called with the pointers a and b pointing to objects that overlap in memory. You are promising that accesses through a and b will never conflict; therefore, a write through one pointer cannot affect a read from any other pointers. The precise semantics of the restrict keyword are described in the 1999 version of the ANSI/ISO C Standard.

Example 5-7. Use of the restrict Type Qualifier With Pointers

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

Example 5-8 illustrates using the restrict keyword when passing arrays to a function. Here, the arrays c and d should not overlap, nor should c and d point to the same array.

Example 5-8. Use of the restrict Type Qualifier With Arrays

```
void func2(int c[restrict], int d[restrict])
{
  int i;

  for(i = 0; i < 64; i++)
  {
    c[i] += d[i];
    d[i] += 1;
  }
}
```

5.5.6 The volatile Keyword

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 must 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 and order of accesses of a volatile variable are exactly as they appear in the C/C++ code, no more and no less.

There are different ways to understand how volatile works, but fundamentally it is a hint to the the compiler that something it cannot understand is going on, and so the compiler should not try to be over-clever.

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.

Volatile 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.

Example 5-9. Volatile for Local Variables With setjmp

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

5.6 C++ Exception Handling

The compiler supports all the C++ exception handling features as defined by the ANSI/ISO 14882 C++ Standard. More details are discussed in *The C++ Programming Language, Third Edition* by Bjarne Stroustrup.

The compiler --exceptions option enables exception handling. The compiler's default is no exception handling support.

For exceptions to work correctly, all C++ files in the application must be compiled with the --exceptions option, regardless of whether exceptions occur in a particular file. Mixing exception-enabled object files and libraries with object files and libraries that do not have exceptions enabled can lead to undefined behavior.

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 --exceptions causes the compiler to insert exception handling code. This code will increase the code size of the programand execution time, even if no exceptions are thrown.

See Section 7.1 for details on the run-time libraries.

5.7 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 6.3.

5.8 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—the *asm* statement. The asm (or __asm) statement provides access to hardware features that C/C++ cannot provide. The asm statement is syntactically like a call to a function named asm, with one string constant argument:

asm(" *assembler text* **");**

The compiler copies the argument string directly into your output file. The assembler text must be enclosed in double quotes. All the usual character string escape codes retain their definitions. For example, you can insert a .byte directive that contains quotes as follows:

```
asm("STR: .byte \"abc\"");
```

The inserted code must be a legal assembly language statement. Like all assembly language statements, the line of code inside the quotes must begin with a label, a blank, a tab, or a comment (asterisk or semicolon). The compiler performs no checking on the string; if there is an error, the assembler detects it. For more information about the assembly language statements, see the *TMS320C55x Assembly Language Tools User's Guide*.

The asm statements do not follow the syntactic restrictions of normal C/C++ statements. Each can appear as a statement or a declaration, even outside of blocks. This is useful for inserting directives at the very beginning of a compiled module.

Use the alternate statement __asm("assembler text") if you are writing code for strict ANSI/ISO C mode (using the --strict_ansi option).

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

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

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

5.9 Pragma Directives

Pragma directives tell the compiler how to treat a certain function, object, or section of code. The C55x C/C++ compiler supports the following pragmas:

- C54X_CALL (See Section 5.9.1)
- C54X_FAR_CALL (See Section 5.9.1)
- CHECK_MISRA (See Section 5.9.2)
- CLINK (See Section 5.9.3)
- CODE_SECTION (See Section 5.9.4)
- DATA_ALIGN (See Section 5.9.5)
- DATA_SECTION (See Section 5.9.6)
- DIAG_SUPPRESS, DIAG_REMARK, DIAG_WARNING, DIAG_ERROR, and DIAG_DEFAULT (See Section 5.9.7)
- FAR—only used for extaddr.h (See Section 5.9.8)
- FUNC_ALWAYS_INLINE (See Section 5.9.9)
- FUNC_CANNOT_INLINE (See Section 5.9.10)
- FUNC_EXT_CALLED (See Section 5.9.11)
- FUNC_IS_PURE (See Section 5.9.12)
- FUNC_IS_SYSTEM (See Section 5.9.13)
- FUNC_NEVER_RETURNS (See Section 5.9.14)
- FUNC_NO_GLOBAL_ASG (See Section 5.9.15)
- FUNC_NO_IND_ASG (See Section 5.9.16)
- FUNCTION_OPTIONS (See Section 5.9.17)
- INTERRUPT (See Section 5.9.18)
- INTR_FUNC (See Section 5.9.19)
- MUST_ITERATE (See Section 5.9.20)
- NO_HOOKS (See Section 5.9.21)
- RESET_MISRA (See Section 5.9.22)
- RETAIN (See Section 5.9.23)
- SET_CODE_SECTION (See Section 5.9.24)
- SET_DATA_SECTION (See Section 5.9.24)
- TRAP_FUNC (SeeSection 5.9.19)
- UNROLL (See Section 5.9.25)

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 the pragmas that apply to functions or symbols (except CLINK and RETAIN), the syntax for the pragmas differs between C and C++. In C, you must supply the name of the object or function to which you are applying the pragma as the first argument. In C++, the name is omitted; the pragma applies to the declaration of the object or function that follows it.

When you mark a function with a pragma, you assert to the compiler that the function meets the pragma's specifications in every circumstance. If the function does not meet these specifications at all times, the compiler's behavior will be unpredictable.

5.9.1 The C54X_CALL and C54X_FAR_CALL Pragmas

The C54X_CALL and C54X_FAR_CALL pragmas provide a method for making calls from C55x C code to C54x assembly functions ported with masm55. These pragmas handle the differences in the C54x and C55x run-time environments.

The syntax of the pragma in C is:

#pragma C54X_CALL (*asm function* **);**

#pragma C54X_FAR_CALL (*asm function* **);**

The syntax of the pragma in C++ is:

#pragma C54X_CALL ;

#pragma C54X_FAR_CALL ;

The function is a C54x assembly function, unmodified for C55x in any way, that is known to work with the C54x C compiler. This pragma cannot be applied to a C function.

The appropriate pragma must appear before any declaration or call to the assembly function. Consequently, it should most likely be specified in a header file.

Use C54X_FAR_CALL only when calling C54x assembly functions that must be called with FCALL.

Use C54X_CALL for any other call to a C54x assembly function. This includes calls that use the __far_mode symbol to decide at assembly time whether to use either CALL or FCALL. Such calls always use CALL on C55x.

When the compiler encounters one of these pragmas, it will:

1. Temporarily adopt the C54x calling conventions and the run-time environment required for ported C54x assembly code. For more information on the C54x run-time environment, see the *TMS320C54x Optimizing C Compiler User's Guide*.
2. Call the specified assembly function.
3. Capture the result.
4. Revert to the native C55x calling conventions and run-time environment.

These pragmas do not provide support for C54x assembly functions that call C code. In this case, you must modify the assembly code to account for the differences in the C54x and C55x run-time environments. For more information, see the *Migrating a C54x System to a C55x System* chapter in the *Assembly Language Tools User's Guide*.

C54x assembly code ported for execution on C55x requires the use of 32-bit stack mode. However, the reset vector in the C55x run-time library (in vectors.asm) sets the stack to be in fast return mode. If you use the C54X_CALL or C54X_FAR_CALL pragma in your code, you must change the reset vector as follows:

1. Extract vectors.asm from rts.src. The rts.src file is located in the lib subdirectory.

   ```
   ar55 -x rts.src vectors.asm
   ```

2. In vectors.asm, change the following line:

   ```
   _Reset:  .ivec _c_int00, USE_RETA
   ```

   to be:

   ```
   _Reset:  .ivec _c_int00, C54X_STK
   ```

3. Re-assemble vectors.asm:

   ```
   cl55 vectors.asm
   ```

4. Place the new file into the object and source libraries.

   ```
   ar55 -r rts55.lib vectors.obj
   ar55 -r rts55x.lib vectors.obj
   ar55 -r rts.src vectors.asm
   ```

These pragmas cannot be used in:

- Indirect calls
- Files compiled for the C55x large memory model (with the --memory_model=large option). Data pointers in the large memory model are 23 bits. When passed on the stack, the pointers occupy two words, which is incompatible with functions that expect these pointers to occupy one word.

The C55x C compiler does not support the global register capability of the C54x C compiler.

5.9.2 The CHECK_MISRA Pragma

The CHECK_MISRA pragma enables/disables MISRA-C:2004 rules at the source level. This pragma is equivalent to using the --check_misra option.

The syntax of the pragma in C is:

#pragma CHECK_MISRA (" {all|required|advisory|none|*rulespec*} **");**

The *rulespec* parameter is a comma-separated list of specifiers. See Section 5.3 for details.

The RESET_MISRA pragma can be used to reset any CHECK_MISRA pragmas; see Section 5.9.22.

5.9.3 The CLINK Pragma

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 indicates to the linker that the section is eligible for removal during conditional linking. Therefore, if the section is not referenced by any other section in the application that is being compiled and linked, it will not be included in the output file result of the link.

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

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

The RETAIN pragma has the opposite effect of the CLINK pragma. See Section 5.9.23 for more details.

5.9.4 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 syntax of the pragma in C is:

#pragma CODE_SECTION (*symbol* **, "***section name* **")**

The syntax of the pragma in C++ is:

#pragma CODE_SECTION (" *section name* **")**

The CODE_SECTION pragma is useful if you have code objects that you want to link into an area separate from the .text section.

The following examples demonstrate the use of the CODE_SECTION pragma.

Example 5-10. Using the CODE_SECTION Pragma C Source File

```
#pragma CODE_SECTION(funcA,"codeA")
int funcA(int a)

{
    int i;
    return (i = a);
}
```

Example 5-11. Generated Assembly Code From Example 5-10

```
       .sect "codeA"
       .global _funcA

;********************************************************
;* FUNCTION NAME: _funcA
;********************************************************
_funcA:
       RET
```

Example 5-12. Using the CODE_SECTION Pragma C++ Source File

```
#pragma CODE_SECTION("codeB")
int i_arg(int x) { return 1; }
int f_arg(float x) { return 2; }
```

Example 5-13. Generated Assembly Code From Example 5-12

```
          .sect "codeB"
_i_arg__Fi:
          MOV #1, T0
          RET
          .sect ".text"
_f_arg__Ff:
          MOV #2, T0
          RET
```

5.9.5 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 syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

5.9.6 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*.

The syntax of the pragma in C is:

#pragma DATA_SECTION (*symbol* **, "** *section name* **");**

The syntax of the pragma in C++ is:

#pragma DATA_SECTION (" *section name* **");**

The DATA_SECTION pragma is useful if you have data objects that you want to link into an area separate from the .bss section.

Example 5-14 through Example 5-16 demonstrate the use of the DATA_SECTION pragma.

Example 5-14. Using the DATA_SECTION Pragma C Source File

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

Example 5-15. Using the DATA_SECTION Pragma C++ Source File

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

Example 5-16. Using the DATA_SECTION Pragma Assembly Source File

```
          .global _bufferA
          .bss _bufferA,512,0,0
          .global _bufferB
_bufferB: .usect "my_sect",512,0,0
```

5.9.7 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

The syntax of the pragmas in C is:

#pragma DIAG__XXX_ [=]*num*[, *num*$_2$, *num*$_3$**...**]

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 diagnostics 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 along with the message when the -pden command line option is specified.

5.9.8 The FAR Pragma

The pragma FAR can be used to obtain the full 23-bit address of a data object declared at file scope. The pragma is used only for the extaddr.h header file and is only available in P2 Reserved Mode.

The syntax of the pragma in C is:

#pragma FAR (*x* **);**

The syntax of the pragma in C++ is:

#pragma FAR ;

The pragma must immediately precede the definition of the symbol to which it applies as shown in Example 5-17.

Example 5-17. Using the FAR Pragma

```
#pragma FAR(x)
int x;        /* ok */

#pragma FAR(z)
int y;        /* error - z's definition must */
int z;        /*  immediately follow #pragma */
```

5.9.9 The FUNC_ALWAYS_INLINE Pragma

The FUNC_ALWAYS_INLINE pragma instructs the compiler to always inline the named function. The compiler only inlines the function if it is legal to inline the function and the compiler is invoked with any level of optimization (--opt_level=0).

The pragma must appear before any declaration or reference to the function that you want to inline. In C, the argument *func* is the name of the function that will be inlined. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

#pragma FUNC_ALWAYS_INLINE (*func* **);**

The syntax of the pragma in C++ is:

#pragma FUNC_ALWAYS_INLINE;

Use Caution with the FUNC_ALWAYS_INLINE Pragma

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

5.9.10 The FUNC_CANNOT_INLINE Pragma

The FUNC_CANNOT_INLINE pragma instructs the compiler that the named function cannot be expanded inline. Any function named with this pragma overrides any inlining you designate in any other way, such as using the inline keyword. Automatic inlining is also overridden with this pragma; see Section 2.10.

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

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma FUNC_CANNOT_INLINE;

5.9.11 The FUNC_EXT_CALLED Pragma

When you use the --program_level_compile option, the compiler uses program-level optimization. When you use this type of optimization, the compiler removes any function that is not called, directly or indirectly, by main. You might have C/C++ functions that are called by hand-coded assembly instead of main.

The FUNC_EXT_CALLED pragma specifies to the optimizer to keep these C functions or any other functions that these C/C++ functions call. These functions act as entry points into C/C++.

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

The syntax of the pragma in C is:

```
#pragma FUNC_EXT_CALLED ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_EXT_CALLED;
```

Except for _c_int00, which is the name reserved for the system reset interrupt for C/C++programs, the name of the interrupt (the *func* argument) does not need to conform to a naming convention.

When you use program-level optimization, you may need to use the FUNC_EXT_CALLED pragma with certain options. See Section 3.3.2.

5.9.12 The FUNC_IS_PURE Pragma

The FUNC_IS_PURE pragma specifies to the compiler that the named function has no side effects. This allows the compiler to do the following:

- Delete the call to the function if the function's value is not needed
- Delete duplicate functions

The pragma must appear before any declaration or reference to the function. In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_IS_PURE ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_IS_PURE;
```

5.9.13 The FUNC_IS_SYSTEM Pragma

The FUNC_IS_SYSTEM pragma specifies to the compiler that the named function has the behavior defined by the ANSI/ISO standard for a function with that name.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function to treat as an ANSI/ISO standard function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma FUNC_IS_SYSTEM;

5.9.14 The FUNC_NEVER_RETURNS Pragma

The FUNC_NEVER_RETURNS pragma specifies to the compiler that the function never returns to its caller.

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

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma FUNC_NEVER_RETURNS;

5.9.15 The FUNC_NO_GLOBAL_ASG Pragma

The FUNC_NO_GLOBAL_ASG pragma specifies to the compiler that the function makes no assignments to named global variables and contains no asm statements.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that makes no assignments. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma FUNC_NO_GLOBAL_ASG;

5.9.16 The FUNC_NO_IND_ASG Pragma

The FUNC_NO_IND_ASG pragma specifies to the compiler that the function makes no assignments through pointers and contains no asm statements.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that makes no assignments. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma FUNC_NO_IND_ASG;

5.9.17 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***");**

5.9.18 The INTERRUPT Pragma

The INTERRUPT pragma enables you to handle interrupts directly with C code. In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma INTERRUPT ;

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

Except for _c_int00, which is the name reserved for the system reset interrupt for C programs, the name of the interrupt (the *func* argument) does not need to conform to a naming convention.

HWI Objects and the INTERRUPT Pragma

NOTE: The INTERRUPT pragma must not be used when BIOS HWI objects are used in conjunction with C functions. The HWI_enter/HWI_exit macros and the HWI dispatcher contain this functionality, and the use of the C modifier can cause negative results.

5.9.19 The INTR_FUNC and TRAP_FUNC Pragmas

The INTR_FUNC and TRAP_FUNC pragmas indicate that a function is to be invoked using the machine instructions intr and trap. 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 pragmas in C is:

#pragma INTR_FUNC (*func* **,** *index* **);**
#pragma TRAP_FUNC (*func* **,** *index* **);**

The syntax of the pragmas in C++ is:

#pragma INTR_FUNC (*index* **);**
#pragma TRAP_FUNC (*index* **);**

The *index* is an integer in the range 0 through 31 used as the argument for the intr or trap instruction and that selects the interrupt vector address to be used to transfer to the function.

The call to and return from the function named in one of the pragmas are synchronous transfers of control. The destination address of the call is found in the interrupt vector designated by the *index* argument to the pragma.

The parameter passing and return value conventions for functions designated by these pragmas are the same as they are when no such pragma is given. In the bodies of these functions addressing of entities allocated on the stack is adjusted to account for the hardware stack conventions. Normally, you do need not be concerned with this. See the CPU Reference Guide for details if interested.

Functions invoked with intr or trap may have other aspects of their calling conventions modified by the pragmas C55x_CALL, C54X_CALL, WORD_MODE_FUNC or by the -call compiler option.

It is your responsibility to initialize the appropriate interrupt vectors to map the indicated index to the desired function. The linkname of the function must be used in the interrupt vector. For C functions this is normally just a "_" character followed by the C identifier. For C++ functions, the name is the mangled function name which can be found by inspecting the generated assembly file (saved by means of the --keep_asm option) or by using nm55 to examine the names in the generated object file.

5.9.20 The MUST_ITERATE Pragma

The MUST_ITERATE pragma specifies to the compiler certain properties of a loop. You guarantee that these properties are always true. Through the use of the MUST_ITERATE pragma, you can guarantee that a loop executes a specific number of times. Anytime the UNROLL pragma is applied to a loop, MUST_ITERATE should be applied to the same loop. For loops the MUST_ITERATE pragma's third argument, multiple, is the most important and should always be specified.

Furthermore, the MUST_ITERATE pragma should be applied to any other loops as often as possible. This is because the information provided via the pragma (especially the minimum number of iterations) aids the compiler in choosing the best loops and loop transformations (that is, software pipelining and nested loop transformations). It also helps the compiler reduce code size.

No statements are allowed between the MUST_ITERATE pragma and the for, while, or do-while loop to which it applies. However, other pragmas, such as UNROLL and PROB_ITERATE, can appear between the MUST_ITERATE pragma and the loop.

5.9.20.1 The MUST_ITERATE Pragma Syntax

The syntax of the pragma for C and C++ is:

#pragma MUST_ITERATE (*min*, *max*, *multiple* **);**

The arguments *min* and *max* are programmer-guaranteed minimum and maximum trip counts. The trip count is the number of times a loop iterates. The trip count of the loop must be evenly divisible by *multiple*. All arguments are optional. For example, if the trip count could be 5 or greater, you can specify the argument list as follows:

```
#pragma MUST_ITERATE(5);
```

However, if the trip count could be any nonzero multiple of 5, the pragma would look like this:

```
#pragma MUST_ITERATE(5, , 5); /* Note the blank field for max */
```

It is sometimes necessary for you to provide min and multiple in order for the compiler to perform unrolling. This is especially the case when the compiler cannot easily determine how many iterations the loop will perform (that is, the loop has a complex exit condition).

When specifying a multiple via the MUST_ITERATE pragma, results of the program are undefined if the trip count is not evenly divisible by multiple. Also, results of the program are undefined if the trip count is less than the minimum or greater than the maximum specified.

If no min is specified, zero is used. If no max is specified, the largest possible number is used. If multiple MUST_ITERATE pragmas are specified for the same loop, the smallest max and largest min are used.

5.9.20.2 Using MUST_ITERATE to Expand Compiler Knowledge of Loops

Through the use of the MUST_ITERATE pragma, you can guarantee that a loop executes a certain number of times. The example below tells the compiler that the loop is guaranteed to run exactly 10 times:

```
#pragma MUST_ITERATE(10,10);

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

In this example, the compiler attempts to generate a software pipelined loop even without the pragma. However, if MUST_ITERATE is not specified for a loop such as this, the compiler generates code to bypass the loop, to account for the possibility of 0 iterations. With the pragma specification, the compiler knows that the loop iterates at least once and can eliminate the loop-bypassing code.

MUST_ITERATE can specify a range for the trip count as well as a factor of the trip count. For example:

```
pragma MUST_ITERATE(8, 48, 8);

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

This example tells the compiler that the loop executes between 8 and 48 times and that the trip_count variable is a multiple of 8 (8, 16, 24, 32, 40, 48). The multiple argument allows the compiler to unroll the loop.

You should also consider using MUST_ITERATE for loops with complicated bounds. In the following example:

```
for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

The compiler would have to generate a divide function call to determine, at run time, the exact number of iterations performed. The compiler will not do this. In this case, using MUST_ITERATE to specify that the loop always executes eight times allows the compiler to attempt to generate a software pipelined loop:

```
#pragma MUST_ITERATE(8, 8);

for(i2 = ipos[2]; i2 < 40; i2 += 5)  { ...
```

5.9.21 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 2.12 for details on entry and exit hooks.

5.9.22 The RESET_MISRA Pragma

The RESET_MISRA pragma resets the specified MISRA-C:2004 rules to the state they were before any CHECK_MISRA pragmas (see Section 5.9.2) were processed. For instance, if a rule was enabled on the command line but disabled in the source, the RESET_MISRA pragma resets it to enabled. This pragma accepts the same format as the --check_misra option, except for the "none" keyword.

The syntax of the pragma in C is:

#pragma RESET_MISRA (" {all|required|advisory|*rulespec*} **")**

The *rulespec* parameter is a comma-separated list of specifiers. See Section 5.3 for details.

5.9.23 The RETAIN Pragma

The RETAIN pragma can be applied to a code or data symbol. It causes a .retain directive to be generated into the section that contains the definition of the symbol. The .retain directive indicates to the linker that the section is ineligible for removal during conditional linking. Therefore, regardless whether or not the section is referenced by another section in the application that is being compiled and linked, it will be included in the output file result of the link.

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

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

The CLINK pragma has the opposite effect of the RETAIN pragma. See Section 5.9.3 for more details.

5.9.24 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 Example 5-18 x and y are put in the section mydata. To reset the current section to the default used by the compiler, a blank paramater 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.

Example 5-18. 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:

Example 5-19. Setting a Section With SET_CODE_SECTION Pragma

```
#pragma SET_CODE_SECTION("func1")
extern void func1();
#pragma SET_CODE_SECTION()
...
void func1() { ... }
```

In Example 5-19 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:

Example 5-20. 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 Example 5-20 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 implictly created objects, such as implicit constructors and virtual function tables.

5.9.25 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 );
```

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.

5.10 The _Pragma Operator

The C55x 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.

5.11 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. This algorithm may *mangle* the name.

The linkname for all objects and functions is the same as the name in the C source with an added underscore prefix. This prevents any C identifier from colliding with any identifier in the assembly code namespace, such as an assembler keyword.

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 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 8 for more information.

5.12 Initializing Static and Global Variables

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

Initialize Global Objects

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

5.12.1 Initializing Static and Global Variables With the Linker

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

```
SECTIONS
   {
      ...

      .bss: {} = 0x00;
      ...
   }
```

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

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

For more information about linker command files and the SECTIONS directive, see the linker description information in the *TMS320C55x Assembly Language Tools User's Guide.*

5.12.2 Initializing Static and Global Variables With the const Type Qualifier

Static and global variables of type *const* without explicit initializations are similar to other static and global variables because they might not be preinitialized to 0 (for the same reasons discussed in Section 5.12). For example:

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

However, the initialization of const global and static variables is different because these variables are declared and initialized in a section called .const. For example:

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

This corresponds to an entry in the .const section:

```
   .sect   .const
_zero
   .word   0
```

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

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

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

is compiled into this assembly code:

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

5.13 Changing the ANSI/ISO C Language Mode

The --kr_compatible, --relaxed_ansi, and --strict_ansi options let you specify how the C/C++ compiler interprets your source code. You can compile your source code in the following modes:

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

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

K&R C mode does not apply to C++ code.

5.13.1 Compatibility With K&R C (--kr_compatible Option)

The ANSI/ISO C/C++ language is a superset of the de facto C standard defined in Kernighan and Ritchie's *The C Programming Language*. Most programs written for other non-ANSI/ISO compilers correctly compile and run without modification.

There are subtle changes, however, in the language that can affect existing code. Appendix C in *The C Programming Language* (second edition, referred to in this manual as K&R) summarizes the differences between ANSI/ISO C and the first edition's C standard (the first edition is referred to in this manual as K&R C).

To simplify the process of compiling existing C programs with the ANSI/ISO C/C++ compiler, the compiler has a K&R option (--kr_compatible) that modifies some semantic rules of the language for compatibility with older code. In general, the --kr_compatible option relaxes requirements that are stricter for ANSI/ISO C than for K&R C. The --kr_compatible option does not disable any new features of the language such as function prototypes, enumerations, initializations, or preprocessor constructs. Instead, --kr_compatible simply liberalizes the ANSI/ISO rules without revoking any of the features.

The specific differences between the ANSI/ISO version of C and the K&R version of C are as follows:

- The integral promotion rules have changed regarding promoting an unsigned type to a wider signed type. Under K&R C, the result type was an unsigned version of the wider type; under ANSI/ISO, the result type is a signed version of the wider type. This affects operations that perform differently when applied to signed or unsigned operands; namely, comparisons, division (and mod), and right shift:

  ```
  unsigned short u;
  int i;
  if (u < i)         /* SIGNED comparison, unless --kr_compatible used */
  ```

- ANSI/ISO prohibits combining two pointers to different types in an operation. In most K&R compilers, this situation produces only a warning. Such cases are still diagnosed when --kr_compatible is used, but with less severity:

  ```
  int *p;
  char *q = p;       /* error without --kr_compatible, warning with --kr_compatible */
  ```

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

  ```
  a;                 /* illegal unless --kr_compatible used */
  ```

- ANSI/ISO interprets file scope definitions that have no initializers as *tentative definitions.* In a single module, multiple definitions of this form are fused together into a single definition. Under K&R, each definition is treated as a separate definition, resulting in multiple definitions of the same object and usually an error. For example:

  ```
  int a;
  int a;              /* illegal if --kr_compatible used, OK if not */
  ```

  Under ANSI/ISO, the result of these two definitions is a single definition for the object a. For most K&R compilers, this sequence is illegal, because int a is defined twice.

- ANSI/ISO prohibits, but K&R allows objects with external linkage to be redeclared as static:

  ```
  extern int a;
  static int a;       /* illegal unless --kr_compatible used */
  ```

- Unrecognized escape sequences in string and character constants are explicitly illegal under ANSI/ISO but ignored under K&R:

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

- ANSI/ISO specifies that bit fields must be of type int or unsigned. With --kr_compatible, bit fields can be legally defined with any integral type. For example:

  ```
  struct s
  {
     short f : 2;     /* illegal unless --kr_compatible used */
  };
  ```

- K&R syntax allows a trailing comma in enumerator lists:

  ```
  enum { a, b, c, }; /* illegal unless --kr_compatible used */
  ```

- K&R syntax allows trailing tokens on preprocessor directives:

  ```
  #endif NAME         /* illegal unless --kr_compatible used */
  ```

5.13.2 Enabling Strict ANSI/ISO Mode and Relaxed ANSI/ISO Mode (--strict_ansi and --relaxed_ansi Options)

Use the --strict_ansi option when you want to compile under strict ANSI/ISO mode. In this mode, error messages are provided when non-ANSI/ISO features are used, and language extensions that could invalidate a strictly conforming program are disabled. Examples of such extensions are the inline and asm keywords.

Use the --relaxed_ansi option when you want the compiler to ignore strict ANSI/ISO violations rather than emit a warning (as occurs in normal ANSI/ISO mode) or an error message (as occurs in strict ANSI/ISO mode). In relaxed ANSI/ISO mode, the compiler accepts extensions to the ANSI/ISO C standard, even when they conflict with ANSI/ISO C. The GCC language extensions described in Section 5.14 are available in relaxed ANSI/ISO mode.

5.13.3 Enabling Embedded C++ Mode (--embedded_cpp Option)

The compiler supports the compilation of embedded C++. In this mode, some features of C++ are removed that are of less value or too expensive to support in an embedded system. When compiling for embedded C++, the compiler generates diagnostics for the use of omitted features.

Embedded C++ is enabled by compiling with the --embedded_cpp option.

Embedded C++ omits these C++ features:

- Templates
- Exception handling
- Run-time type information
- The new cast syntax
- The keyword mutable
- Multiple inheritance
- Virtual inheritance

Under the standard definition of embedded C++, namespaces and using-declarations are not supported. The C55x compiler nevertheless allows these features under embedded C++ because the C++ run-time-support library makes use of them. Furthermore, these features impose no run-time penalty.

The compiler does not support embedded C++ run-time-support libraries.

5.14 GNU 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 3.4) can be found at the GNU web site, http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/C-Extensions.html.

Most of these extensions are also available for C++ source code.

5.14.1 Extensions

Most of the GCC language extensions are available in the TI compiler when compiling in relaxed ANSI mode (--relaxed_ansi) or if the --gcc option is used.

The extensions that the TI compiler supports are listed in Table 5-2, which is based on the list of extensions found at the GNU web site. The shaded rows describe extensions that are not supported.

Table 5-2. 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
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.

[(1)] Feature defined for GCC 3.0; definition and examples at http://gcc.gnu.org/onlinedocs/gcc-3.0.4/gcc/C-Extensions.html

Table 5-2. GCC Language Extensions (continued)

Extensions	Descriptions
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
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 5.14.5)
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

5.14.2 Function Attributes

The following GCC function attributes are supported: always_inline, const, constructor, deprecated, format, format_arg, malloc, noinline, noreturn, pure, section, unused, used and warn_unused_result.

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.

The malloc attribute is applied to the declarations of malloc, calloc, realloc and memalign in stdlib.h.

5.14.3 Variable Attributes

The following variable attributes are supported: aligned, deprecated, mode, packed, section, transparent_union, unused, and used.

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).

5.14.4 Type Attributes

The following type attributes are supported: aligned, deprecated, packed, transparent_union, and unused.

Members of a packed structure are stored as closely to each other as possible, omitting additional bytes of padding usually added to preserve word-alignment. For example, assuming a word-size of 4 bytes ordinarily has 3 bytes of padding between members c1 and i, and another 3 bytes of trailing padding after member c2, leading to a total size of 12 bytes:

```
struct unpacked_struct { char c1; int i; char c2;};
```

However, the members of a packed struct are byte-aligned. Thus the following does not have any bytes of padding between or after members and totals 6 bytes:

```
struct __attribute__((__packed__)) packed_struct { char c1; int i; char c2; };
```

Subsequently, packed structures in an array are packed together without trailing padding between array elements.

Bit fields of a packed structure are bit-aligned. The byte alignment of adjacent struct members that are not bit fields does not change. However, there are no bits of padding between adjacent bit fields.

The packed attribute can only be applied to the original definition of a structure or union type. It cannot be applied with a typedef to a non-packed structure that has already been defined, nor can it be applied to the declaration of a struct or union object. Therefore, any given structure or union type can only be packed or non-packed, and all objects of that type will inherit its packed or non-packed attribute.

The packed attribute is not applied recursively to structure types that are contained within a packed structure. Thus, in the following example the member s retains the same internal layout as in the first example above. There is no padding between c and s, so s falls on an unaligned boundary:

```
struct __attribute__((__packed__)) outer_packed_struct { char c; struct unpacked_struct s; };
```

It is illegal to implicitly or explicitly cast the address of a packed struct member as a pointer to any non-packed type except an unsigned char. In the following example, p1, p2, and the call to foo are all illegal.

```
void foo(int *param);
struct packed_struct ps;

int *p1 = &ps.i;
int *p2 = (int *)&ps.i;
foo(&ps.i);
```

However, it is legal to explicitly cast the address of a packed struct member as a pointer to an unsigned char:

```
unsigned char *pc = (unsigned char *)&ps.i;
```

Packed can also be applied to enumerated types. On an enum, packed indicates that the smallest integral type should be used.

The TI compiler also supports an unpacked attribute for an enumeration type to allow you to indicate that the representation is to be an integer type that is no smaller than int; in other words, it is not *packed*.

5.14.5 Built-In Functions

The following builtin functions are supported: __builtin_abs, __builtin_classify_type, __builtin_constant_p, __builtin_expect, __builtin_fabs, __builtin_fabsf, __builtin_frame_address, __builtin_labs, __builtin_llabs, __builtin_memcpy, and __builtin_return_address.

The __builtin_frame_address function always returns zero.

The __builtin_return_address function always returns zero.

5.15 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.

Run-Time Environment

This chapter describes the TMS320C55x C/C++ run-time environment. To ensure successful execution of C/C++ programs, it is critical that all run-time code maintain this environment. It is also important to follow the guidelines in this chapter if you write assembly language functions that interface with C/C++ code.

Topic **Page**

6.1 Memory

The C55x compiler treats memory as a single linear block that is partitioned into subblocks of code and data. Each subblock of code or data generated by a C program is placed in its own continuous memory space. The compiler assumes that a full 24-bit address space is available in target memory.

The C55x compiler supports a small memory model, a large memory model, and a huge memory model. These memory models affect how data is placed in memory and accessed.

All the code in an application must use the same memory model. The linker does not allow code compiled for different memory models to be linked together. The appropriate run-time library must be used for each memory model. See Section 7.1.6 for library naming conventions.

6.1.1 Small Memory Model

The use of the small memory model results in code and data sizes that are slightly smaller and more efficient than when using the larger memory models. However, your program must meet certain size and memory placement restrictions.

In the small memory model, the following sections must all fit within a single page of memory that is 64K words in size.

- The .bss and .data sections (all static and global data)
- The .stack and .sysstack sections (the primary and secondary system stacks)
- The .sysmem section (dynamic memory space)
- The .const section

No section may cross a hardware page boundary. There is no other restriction on the size or placement of .text sections (code), .switch sections (switch statements), or .cinit/.pinit sections (variable initialization).

In the small model, the compiler uses 16-bit data pointers to access data. The upper 7 bits of the XARn registers are set to point to the page that contains the .bss section. They remain set to that value throughout program execution.

6.1.2 Large Memory Model

The large memory model supports fewer restrictions on the placement of data. To use the large memory model, use the --memory_model=large option.

In the large model:

- Data pointers are 23 bits and occupy two words when stored in memory.
- The .stack and .sysstack sections must be on the same page.
- Objects must be 64K words or less.
- For C55x Revision 3 only: Objects may cross page boundaries.

6.1.3 Huge Memory Model

The huge memory model (--memory_model=huge) is available on hardware based on C55x CPU revision 3. Huge model is like large memory model except that objects can be as large as desired, up to the 8MB limit of memory space. In both C and assembly code, you can use the predefined macro __HUGE_MODEL__ to conditionally compile code depending on the model. However, you should not need to use __HUGE_MODEL__ in C code with the proper use of the size_t and ptrdiff_t types.

The performance of huge model is similar to that of large model; however, there is one case of inefficiency you might notice. Since the C55x zero-overhead looping construct (RPT, RPTB) allows only a 16-bit loop count, loops that iterate over size_t values are done with the less efficient conditional branch. However, calls to the library string functions that can be inlined (such as, memcpy) with a constant count less than 16 bits are done with RPT.

Object files generated in the huge model are incompatible with C55x CPU revisions earlier than 3.0. The compiler and will enforce using C55x CPU revision 3.0 for huge model.

The compiler and prevent object files optimized for CPU revisions 1 and 2 only from being combined with object files optimized for CPU revision 3 only.

Missing Feature of the Huge Memory Model

NOTE: The dynamic memory allocation system has not yet been updated to take advantage of the more flexible object size; this will be addressed in a future release of the tools.

The Linker Defines the Memory Map

NOTE: 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).

6.1.4 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 *TMS320C55x Assembly Language Tools User's Guide*.

There are two basic types of sections:

- **Initialized sections** contain data or executable code. The C/C++ compiler creates the following initialized sections:
  - The **.cinit section** contains tables for initializing variables and constants.
  - The **.pinit section** contains tables for global object constructors at run time.
  - The **.const section** contains string constants and data defined with the C/C++ qualifier *const* (provided the constant is not also defined as *volatile*).
  - The **.switch section** contains tables for switch statements.
  - The **.text section** contains all the executable code.
- **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at run time to create and store variables. The compiler creates the following uninitialized sections:
  - The **.bss section** reserves space for uninitialized global and static variables that are within reach of the GDP register offset.
  - The **.stack section** reserves memory for the system stack. This memory passes variables and is used for local storage.
  - 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.
  - The **.cio section** supports C I/O. This space is used as a buffer with the label __CIOBUF_. When any type of C I/O is performed (printf, scanf, etc.), the buffer is created. It contains an internal C I/O command (and required parameters) for the type of stream I/O that is to be performed, as well as the data being returned from the C I/O command. The .cio section must be allocated in the linker command file in order to use C I/O.

Only code sections and large or huge model data sections on C55x revision 3 hardware are allowed to cross page boundaries. Any other type of section must fit on one page.

The assembler creates the default sections .text, .bss, and .data. The C/C++ compiler, however, does not use the .data section. You can instruct the compiler to create additional sections by using the CODE_SECTION and DATA_SECTION pragmas (see Section 5.9.4 and Section 5.9.6).

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 6-1. You can place these output sections anywhere in the address space as needed to meet system requirements.

Table 6-1. Summary of Sections and Memory Placement

Section	Type of Memory	Section	Type of Memory
.args	ROM or RAM	.pinit	ROM or RAM
.bss	RAM	.stack	RAM
.cinit	ROM or RAM	.sysmem	RAM
.cio	RAM	.sysstack	RAM
.const	ROM or RAM	.text	ROM or RAM
.data	ROM or RAM		

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 *TMS320C55x Assembly Language Tools User's Guide*.

6.1.5 C/C++ System Stack

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

- Allocate local variables
- Pass arguments to functions
- Save the processor status

The run-time stack is allocated in a single continuous block of memory and grows down from high addresses to lower addresses. The compiler uses the hardware stack pointer (SP) to manage the stack.

C55x also supports a secondary system stack. For compatibility with C54x, the primary run-time stack holds the lower 16 bits of addresses. The secondary system stack holds the upper 8 bits of C55x return addresses. The compiler uses the secondary stack pointer (SSP) to manage the secondary system stack.

The size of both stacks is set by the linker. The linker also creates global symbols, __STACK_SIZE and __SYSSTACK_SIZE, and assigns them a value equal to the respective sizes of the stacks in bytes. The default stack size is 1000 bytes. The default secondary system stack size is also 1000 bytes. You can change the size of either stack at link time by using the -stack or -sysstack options on the linker command line and specifying the size as a constant immediately after the option.

Placement of .stack and .sysstack Sections

NOTE: The .stack and .sysstack sections must be on the same page for C55x hardware.

Stack Overflow

NOTE: 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 2.12.

6.1.6 Dynamic Memory Allocation

The run-time-support library supplied with the C55x compiler contains several functions (such as malloc, calloc, and realloc) that allow you to allocate memory dynamically for variables at run time.

Memory is allocated from a global pool, or heap, that is defined in the .sysmem section. You can set the size of the .sysmem section by using the --heap_size=*size* option with the linker command. The linker also creates a global symbol, __SYSMEM_SIZE, and assigns it a value equal to the size of the heap in bytes. The default size is 2000 bytes. For more information on the --heap_size option, see the linker description chapter in the *TMS320C55x Assembly Language Tools User's Guide*.

Dynamically allocated objects are not addressed directly (they are always accessed with pointers) and the memory pool is in a separate section (.sysmem); therefore, the dynamic memory pool can have a size limited only by the amount of available memory in your system. To conserve space in the .bss section, you can allocate large arrays from the heap instead of defining them as global or static. For example, instead of a definition such as:

```
struct big table[100];
```

Use a pointer and call the malloc function:

```
struct big *table
table = (struct big *)malloc(100*sizeof(struct big));
```

6.1.7 Initialization of Variables

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

In situations where a program is loaded directly from an object file into memory and run, you can avoid having the .cinit section occupy space in memory. A loader can read the initialization tables directly from the object file (instead of from ROM) and perform the initialization directly at load time instead of at run time. You can specify this to the linker by using the --ram_model link option. For more information, see Section 6.9.

6.1.8 Bit-Field Allocation

Each bit-field has a declared type (int, unsigned int, long or unsigned long for C55x) and a size. (In C the size of a bit-field is never greater than the size of the declared type.) The C standard says that an implementation may allocate the bit-field in any addressable storage unit large enough to hold the bit-field. This *addressable storage unit* is usually called the *container type*. In the C55x implementation, the container type is always the declared type of the bit-field. Note that containers always obey the usual alignment constraints: 8-bit alignment for 8-bit types, 16-bit alignment for 16-bit types, etc.

In the abstract model, access to a bit-field is made by fetching the appropriate container, reading or writing the appropriate bits and storing the container if necessary. Obviously, all such fetches and stores are properly aligned.

To lay out a bit-field, first determine the *current container*. This is the container-sized-and-aligned piece of memory holding the next bit available in the memory used for the structure. For the first bit-field this is the MSB of the first word of the structure's memory. For later bit-fields it is the MSB following the preceding member. If the size of the bit-field is less than or equal to the number of bits remaining in the current container, it is allocated in the current container, otherwise it is moved to the next (aligned) container and allocated there.

Here are some examples.

```
    struct S1 { long a: 8; int b: 8; }
    +-----------------------------------+
    |   0    |   1    |   2    |   3    |
    |aaaaaaaa|bbbbbbbb|........|........|
    +-----------------------------------+
```

Current container when allocating b is the int made up of bytes 0-1.

```
struct S2 { long a: 8; int b: 16; }
+-----------------------------------+
|   0    |   1    |   2    |   3    |
|aaaaaaaa|........|bbbbbbbb|bbbbbbbb|
+-----------------------------------+
```

Current container when allocating b is the int made up of bytes 0-1.

```
struct S3 { long a: 8; long b: 16; }
+-----------------------------------+
|   0    |   1    |   2    |   3    |
|aaaaaaaa|bbbbbbbb|bbbbbbbb|........|
+-----------------------------------+
```

Current container when allocating b is the long made up of bytes 0-3.

This scheme has the interesting side effect that the bigger the declared types of the bit-fields, the less likely that extra intervening padding bits will have to be added.

Even though in the abstract model the fetch/store to access a bit-field is of the size determined by the container type, this will not necessarily be the case in practice. If the compiler knows a better and safer way to do the access, it will. In the abstract model, to set that bit you would fetch a double word, then set the bit and store the double word. In practice, you usually just set the bit using the instruction that sets a bit in memory.

6.2 Object Representation

This section explains how various data objects are sized, aligned, and accessed.

6.2.1 Data Type Storage

lists register and memory storage for various data types:

Table 6-2. Data Representation in Registers and Memory

Data Type	Memory Storage
char, signed char	8 bits aligned to 8-bit boundary
unsigned char, bool	8 bits aligned to 8-bit boundary
short, signed short	16 bits aligned to 16-bit (word) boundary
unsigned short, wchar_t	16 bits aligned to 16-bit (word) boundary
int, signed int	16 bits aligned to 16-bit (word) boundary
unsigned int	16 bits aligned to 16-bit (word) boundary
enum	16 bits aligned to 16-bit (word) boundary
long. signed long	32 bits aligned to 16-bit (word) boundary
unsigned long	32 bits aligned to 16-bit (word) boundary
float	32 bits aligned to 16-bit (word) boundary
double	32 bits aligned to 16-bit (word) boundary
long double	32 bits aligned to 16-bit (word) boundary
struct	Members are stored as their individual types require; aligned according to the member with the most restrictive alignment requirement.
array	Members are stored as their individual types require; aligned to 32-bit (word) boundary. All arrays inside a structure are aligned according to the type of each element in the array.
pointer to data member	16 bits aligned to 16-bit (word) boundary
pointer to function	16 bits aligned to 16-bit (word) boundary

6.2.1.1 Pointer to Member Function Types

Pointer to member function objects are stored as a structure with three members, and the layout is equivalent to:

```
struct {
   short int d;
   short int i;
   union {
      void (f) ();
      long 0; }
   };
```

The parameter d is the offset to be added to the beginning of the class object for this pointer. The parameter I is the index into the virtual function table, offset by 1. The index enables the NULL pointer to be represented. Its value is -1 if the function is nonvirtual. The parameter f is the pointer to the member function if it is nonvirtual, when I is 0. The 0 is the offset to the virtual function pointer within the class object.

6.2.1.2 Structure and Array Alignment

Structures are aligned according to the member with the most restrictive alignment requirement. Structures do not contain padding after the last member. Arrays are always word aligned. Elements of arrays are stored in the same manner as if they were individual objects.

6.2.1.3 Field/Structure Alignment

When the compiler allocates space for a structure, it allocates as many words as are needed to hold all of the structure's members and to comply with alignment constraints for each member.

When a structure contains a 32-bit (long) member, the long is aligned to a 2-word (32-bit) boundary. This may require padding before, inside, or at the end of the structure to ensure that the long is aligned accordingly and that the sizeof value for the structure is an even value.

Fields are packed as they are encountered; the most significant bits of the structure word are filled first.

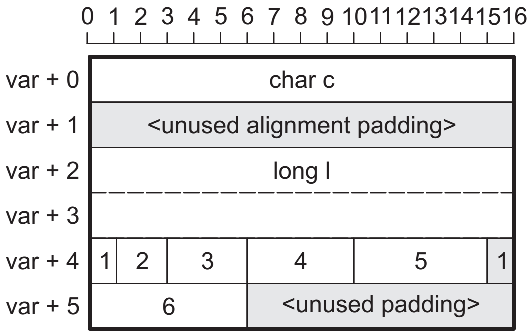
All non-bit-field types are aligned on word boundaries. Bit-fields are allocated as many bits as requested. Adjacent fields are packed into adjacent bits of a word, but they do not overlap words. If a field would overlap into the next word, the entire field is placed into the next word.

Example 6-1 and Figure 6-1 illustrate the C code and memory layout of a structure for C55x.

Example 6-1. C Code Definition of "var"

```
struct example {
    char c;
    long l;
    int bf1:1;
    int bf2:2;
    int bf3:3;
    int bf4:4;
    int bf5:5;
    int bf6:6;
};
```

Figure 6-1. Memory Layout of “var”



6.2.2 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 6.9.

- In an expression. For example:

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

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

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

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

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

  Because strings are stored in the .const section (possibly in ROM) and shared, it is bad practice for a program to modify a string constant. The following code is an example of incorrect string use:

  ```
  const char   *a = "abc"
  a[1] = 'x';             /* Incorrect! */
  ```

6.3 Register Conventions

Strict conventions associate specific registers with specific operations in the C/C++ environment. If you plan to interface an assembly language routine to a C/C++ program, you must understand and follow these register conventions.

6.3.1 General Registers

The register conventions dictate how the compiler uses registers and how values are preserved across function calls. Table 6-3 describes how the registers are used and how they are preserved. The parent function is the function making the function call. The child function is the function being called. For more information about how values are preserved across calls, see Section 6.4.

Registers not used by compiled code are not listed in the table. Registers are used in compiled code at your discretion if their use is not specified in the table.

Table 6-3. Register Use and Preservation Conventions

Register	Preserved By	Uses
AC[0-3]	Parent	16-bit, 32-bit, or 40-bit data, or 24-bit code pointers
(X)AR[0-4]	Parent	16-bit or 23-bit pointers, or 16-bit data
(X)AR[5-7]	Child	16-bit or 23-bit pointers, or 16-bit data
BK03, BK47, BKC	Parent	--
BRC0, BRC1	Parent	--
BRS1	Parent	--
BSA01, BSA23, BSA45, BSA67, BSAC	Parent	--
(X)CDP	Parent	
[1]CFCT	Child	--
CSR	Parent	--
(X)DP	Child	[1] Unused in large memory model
MDP, MDP05, MDP67	Child (in P2 Reserved mode)	--
PC	N/A	--
REA0, REA1	Parent	--
RETA	Child	--
RPTC	Parent	--
RSA0, RSA1	Parent	--
SP	N/A[2]	--
SSP	N/A	--
ST[0-3]_55	See Section 6.3.2	--
T0, T1	Parent	16-bit data
T2, T3	Child	16-bit data
TRN0, TRN1	Parent	--

[1] In small memory model, all extension bits of addressing registers (XARn, XCDP, XDP) are child-saved and presumed to contain the page number of the page containing the program data.

[2] The SP is preserved by the convention that everything pushed on the stack is popped off before returning.

6.3.2 Status Registers

Table 6-4 through Table 6-7 show the status register fields.

The Presumed Value column contains the value that meets one of these conditions:

- The compiler expects in that field upon entry to, or return from, a function.
- An assembly function can expect when the function is called from C/C++ code.
- An assembly function must set upon returning to, or making a call into, C/C++ code. If this is not possible, the assembly function cannot be used with C/C++ functions.

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.

The run-time initialization code (boot.asm) sets the assumed values. For more information on boot.asm, see Section 6.9.

Table 6-4. Status Register Field ST0_55

Field	Name	Presumed Value	Modified
ACOV[0-3]	Overflow detection	-	Yes
CARRY	Carry	-	Yes
TC[1-2]	Test control	-	Yes
DP[07-15]	Data page register	-	No

Table 6-5. Status Register Field ST1_55

Field	Name	Presumed Value	Modified
BRAF	Block-repeat active flag	-	No
CPL	Compiler mode	1	No
XF	External flag	-	No
HM	Hold mode	-	No
INTM	Interrupt mode	-	No
M40	Computation mode (D unit)	0	Yes[1]
SATD	Saturation mode (D unit)	0	Yes
SXMD	Sign-extension mode (D unit)	1	No
C16	Dual 16-bit arithmetic mode	0	No
FRCT	Fractional mode	0	Yes
54CM	C54x compatibility mode	0	Yes[2]
ASM	Accumulator shift mode	-	No

[1] When 40-bit arithmetic is used (long long data)

[2] When the C54X_CALL pragma is used

Table 6-6. Status Register Field ST2_55

Field	Name	Presumed Value	Modified
ARMS	AR mode	1	No
DBGM	Debug enable mask	-	No
EALLOW	Emulation access enable	-	No
RDM	Rounding mode	0	Yes
CDPLC	CDP linear/circular configuration	0	Yes
AR[0-7]LC	AR[0-7] linear/circular configuration	0	Yes

Table 6-7. Status Register Field ST3_55

Field	Name	Presumed Value	Modified
CAFRZ	Cache freeze	-	No
CAEN	Cache enable	-	No
CACLR	Cache clear	-	No
HINT	Host interrupt	-	No
CBERR	CPU bus error flag	-	No
MPNMC	Microprocessor/microcomputer mode	-	No
SATA	Saturate mode (A unit)	0	Yes
CLKOFF	CLKOUT disable	-	No
SMUL	Saturation-on-multiplication mode	1	Yes
SST	Saturation-on-store mode	-	No

6.4 Function Structure and Calling Conventions

The C/C++ compiler imposes a strict set of rules on function calls. Except for special run-time support functions, any function that calls or is called by a C/C++ function must follow these rules. Failure to adhere to these rules can disrupt the C/C++ environment and cause a program to fail.

6.4.1 How a Function Makes a Call

A function (parent function) performs the following tasks when it calls another function (child function). A C routine should be accessed with a call (CALL) instruction and never with a branch (B) instruction.

1. Arguments to a function are placed in registers or on the stack.

   (a) For a function declared with an ellipsis (indicating that it is called with varying numbers of arguments), the last explicitly-declared argument is passed on the stack, followed by the rest of the arguments. Its stack address can act as a reference for accessing the undeclared arguments.

   Arguments declared before the last explicit argument follow rules shown below.

   (b) In general, when an argument is passed to a function, the compiler assigns to it a particular class. It then places it in a register (if available) according to its class. The compiler uses three classes:

   - 24-bit data pointer (int *, long *, etc.)
   - 16-bit data (char, short, int)
   - 32-bit data (long, float, double, function pointers) or 40-bit (long long)

   If the argument is a pointer to any type of data, it is considered a data pointer. If an argument will fit into a 16-bit register, it is considered 16-bit data. Otherwise, it is considered 32-bit data.40-bit data is passed by using the entire register rather than just the lower 32 bits.

   (c) A structure of two words (32 bits) or less is treated like a 32-bit data argument. It is passed in a register, if available.

   (d) A structure larger than two words is passed by reference. The compiler will pass the address of the structure as a pointer. This pointer is treated like a data pointer argument.

   (e) If the called (child) function returns a struct or union value, the parent function allocates space on the local stack for a structure of that size. The parent then passes the address of that space as a hidden first argument to the called function. This pointer is treated like a data pointer argument. In effect, the function call is transformed from:

   ```
   struct s result = fn(x, y);
   ```

   to

   ```
   fn(&result, x, y);
   ```

(f) The arguments are assigned to registers in the order that the arguments are listed in the prototype. They are placed in the following registers according to their class, in the order shown below. For example, the first 32-bit data argument is placed in AC0. The second 32-bit data argument is placed in AC1, and so on.

Argument Class	Assigned to Register(s)
data pointer (24 bits)	(X)AR0, (X)AR1, (X)AR2, (X)AR3, (X)AR4
16-bit data	T0, T1, AR0, AR1, AR2, AR3, AR4
32-bit data or 40-bit data	AC0, AC1, AC2

The ARx registers overlap for data pointers and 16-bit data. If, for example, T0 and T1 hold 16-bit data arguments, and AR0 already holds a data pointer argument, a third 16-bit data argument would be placed in AR1. For an example, see the second prototype in Figure 6-2.

If there are no available registers of the appropriate type, the argument goes on the stack.

(g) Arguments passed on the stack are handled as follows. The stack is initially aligned to an even boundary. Then, each argument is aligned on the stack as appropriate for the argument's type (even alignment for long, long long, float, double, code pointers, and large model data pointers; no alignment for int, short, char, ioport pointers, and small model data pointers). Padding is inserted as necessary to bring arguments to the correct alignment.

2. The child function will save all of the save-on-entry registers (T2, T3, AR5-AR7). However, the parent function must save the values of the other registers, if they are needed after the call, by pushing the values onto the stack.
3. The parent calls the function.
4. The parent collects the return value.
   - Short data values are returned in T0.
   - Long data values are returned in AC0.
   - Data pointer values are returned in (X)AR0.
   - If the child returns a structure, the structure is on the local stack in the space allocated as described in step 1.

Examples of the register argument conventions are shown in Figure 6-2. The registers in the examples are assigned according to the rules specified in steps 1 through 4 above.

Figure 6-2. Register Argument Conventions

```
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);
```

6.4.2 How a Called Function Responds

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

1. The called function allocates enough space on the stack for any local variables, temporary storage areas, and arguments to functions that this function might call. This allocation occurs once at the beginning of the function.
2. If the child function modifies a save-on-entry register (T2, T3, and AR5-AR7), it must save the value, either to the stack or to an unused register. The called function can modify any other register (ACx, Tx, ARx) without saving the value.
3. If the child 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. The local copy of the structure must be created from the passed pointer. If no writes are made to the structure, it can be referenced in the called function indirectly through the pointer argument.
4. The child function executes the code for the function.
5. If the child function returns a value, it places the value accordingly: long data values in AC0, short data values in T0, or data pointers in (X)AR0.

   and passes a pointer to this space in (X)AR0. To return the structure, the called function copies the structure to the memory block pointed to by the extra argument.

   If the parent does not use the return structure value, an address value of 0 can be passed in (X)AR0. This directs the child 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 the caller properly sets up the first argument) and where they are defined (so the function knows to copy the result).

6. The child function restores all registers saved in step 2.
7. The child function restores the stack to its original value.
8. The function returns.

6.4.3 *Accessing Arguments and Local Variables*

The compiler uses the compiler mode (selected when the CPL bit in status register ST1_55 is set to 1) for accessing arguments and locals. When this bit is set, the direct addressing mode computes the data address by adding the constant in the dma field of the instruction to the SP. For example:

```
MOV *SP(#8), T0
```

The largest offset available with this addressing mode is 128. So, if an object is too far away from the SP to use this mode of access, the compiler copies the SP to AR6 (FP) in the function prolog, then uses long offset addressing to access the data. For example:

```
MOV SP, FP
...
MOV *FP(#130), AR3
```

6.5 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 6.5.1).
- Use assembly language variables and constants in C/C++ source (see Section 6.5.2).
- Use inline assembly language embedded directly in the C/C++ source (see Section 6.5.4).
- Use intrinsics in C/C++ source to directly call an assembly language statement (see Section 6.5.5).

6.5.1 *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 6.4, and the register conventions defined in Section 6.3. C/C++ code can access variables and call functions defined in assembly language, and assembly code can access C/C++ variables and call C/C++ functions.

Follow these guidelines to interface assembly language and C:

- All functions, whether they are written in C/C++ or assembly language, must follow the register conventions outlined in Section 6.3.
- You must preserve any dedicated registers modified by a function. Dedicated registers include:
  - Child-saved-entry registers (T2, T3, AR5, AR6, AR7)
  - Stack pointer (SP)

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

  Any register that is not dedicated can be used freely without first being saved.
- To call a C54x assembly function from compiled C55x code, use the C54X_CALL or C54X_FAR_CALL pragma. For more information, see Section 5.9.1.
- Interrupt routines must save *all* the registers they use. For more information, see Section 6.6.
- When calling C/C++ functions, remember that only the dedicated registers are preserved. C/C++ functions can change the contents of any other register.

- The compiler assumes that the stack is initialized at run time to an even word address. If your assembly function calls a C/C++ function, you must align the SP by subtracting an odd number from the SP. The space must then be deallocated at the end of the function by adding the same number to the SP. For example:

  ```
  _func: AADD #-1, SP   ; aligns SP
         ...            ; body of function
         AADD #1, SP    ; deallocate stack space
         RET            ; return from asm function

     MOV *(#global_var),AR3 ; works with CPL = = 1
     MOV global_var, AR3    ; does not work with CPL ==1
  ```

- Longs and floats are stored in memory with the most significant word at the lower address.
- Functions must return values as described in Section 6.4.2.
- 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 places an underscore (_) at the beginning of all identifiers. This name space is reserved by the compiler. Prefix the names of variables and functions that are accessible from C/C++ with _. For example, a C/C++ variable called x is called _x in assembly language.

  For identifiers that are to be used only in an assembly language module or modules, the identifier should not begin with an underscore.
- The compiler assigns linknames to all external objects. Thus, when you are writing assembly language code, you must use the same linknames as those assigned by the compiler. See Section 5.11 for more information.
- Any object or function declared in assembly language that is accessed or called from C/C++ must be declared with the .def or .global directive in the assembly language modifier. This declares the symbol as external and allows the linker to resolve references to it.

  Likewise, to access a C/C++ function or object from assembly language, declare the C/C++ object with the .ref or .global directive in the assembly language module. This creates an undeclared external reference that the linker resolves.
- Because compiled code runs with the CPL (compiler mode) bit set to 1, the only way to access directly addressed objects is with indirect absolute mode. For example:

  ```
  MOV *(#global_var),AR3 ; works with CPL = = 1 MOV global_var, AR3 ; does not work with CPL ==1
  ```

  If you set the CPL bit to 0 in your assembly language function, you must set it back to 1 before returning to compiled code.

Example 6-2 illustrates a C++ function called main, which calls an assembly language function called asmfunc, Example 6-4. The asmfunc function takes its single argument, adds it to the C++ global variable called gvar, and returns the result.

Example 6-2. Calling an Assembly Language Function From a C/C++ Program

```
extern void asmfunc(int *);
int global;

void func()
{

    int local = 5;
    asmfunc(&local);
}
```

Example 6-3. Assembly Language Output of Example 6-2

```
_func:
         AADD #-1, SP
         MOV XSP, XAR0
         MOV #5, *SP(#0)
         CALL #_asmfunc
         AADD #1, SP
         RET
```

Example 6-4. Assembly Language Program Called by Example 6-2

```
_asmfunc:
         MOV *AR0, AR1
         ADD *(#_global), AR1, AR1
         MOV AR1, *(#_global)
         RET
```

In the assembly language code in Example 6-4, note the underscore on the C/C++ symbol name used in the assembly code.

6.5.2 Accessing Assembly Language Variables From C/C++

It is sometimes useful for a C/C++ program to access variables or constants defined in assembly language. There are several methods that you can use to accomplish this, depending on where and how the item is defined: a variable defined in the .bss section, a variable not defined in the .bss section, or a constant.

6.5.2.1 Accessing Assembly Language Global Variables

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

1. Use the .bss or .usect directive to define the variable.
2. 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 6-6 and Example 6-5 show how you can access a variable defined in .bss.

Example 6-5. Assembly Language Variable Program

```
* Note the use of underscores in the following lines

       .bss     _var,1   ; Define the variable
       .global  _var     ; Declare it as external
```

Example 6-6. C Program to Access Assembly Language From Example 6-5

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

You may not always want a variable to be in the .bss section. For example, a common situation is a lookup table defined in assembly language that you do not want to put in RAM. In this case, you must define a pointer to the object and access it indirectly from C/C++.

The first step is to define the object; it is helpful (but not necessary) to put it in its own initialized section. Declare a global label that points to the beginning of the object, and then the object can be linked anywhere into the memory space. To access it in C/C++, you must declare the object as extern and not precede it with an underscore. Then you can access the object normally.

Example 6-7 and Example 6-8 show an example that accesses a variable that is not defined in .bss.

Example 6-7. Accessing from C a Variable Not Defined in .bss

```
extern float sine[];   /* This is the object            */
float *sine_p = sine;  /* Declare pointer to point to it */
f = sine_p[4];         /* Access sine as normal array    */
```

Example 6-8. Assembly Language Program for Example 6-7

```
       .global  _sine      ; Declare variable as external
       .sect    "sine_tab" ; Make a separate section
_sine:                     ; The table starts here
       .float   0.0
       .float   0.015987
       .float   0.022145
```

6.5.2.2 Accessing Assembly Language Constants

You can define global constants in assembly language by using the .set, .def, and .global directives, or you can define them in a linker command file using a linker assignment statement. These constants are accessible from C/C++ only with the use of special operators.

For normal variables defined in C/C++ or assembly language, the symbol table contains the *address of the value* of the variable. For assembler constants, however, the symbol table contains the *value* of the constant. The compiler cannot tell which items in the symbol table are values and which are addresses.

If you try to access an assembler (or linker) constant by name, the compiler attempts to fetch a value from the address represented in the symbol table. To prevent this unwanted fetch, you must use the & (address of) operator to get the value. In other words, if x is an assembly language constant, its value in C/C++ is &x.

You can use casts and #defines to ease the use of these symbols in your program, as in Example 6-9 and Example 6-10.

Example 6-9. 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 6-10. Assembly Language Program for Example 6-9

```
_table_size  .set    10000     ; define the constant
             .global _table_size  ; make it global
```

Because you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example 6-9, int is used. You can reference linker-defined symbols in a similar manner.

6.5.3 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 *TMS320C55x Assembly Language Tools User's Guide*.

6.5.4 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 5.8.

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.

6.5.5 Using Intrinsics to Access Assembly Language Statements

The C55x compiler recognizes a number of intrinsic operators. Intrinsics allow you to express the meaning of certain assembly statements that would otherwise be cumbersome or inexpressible in C/C++. Intrinsics are used like functions; you can use C/C++ variables with these intrinsics, just as you would with any normal function.

The intrinsics are specified with a leading underscore, and are accessed by calling them as you do a function. For example:

```
int x1, x2, y;
y = _sadd(x1, x2);
```

Many of the intrinsic operators support saturation. During saturating arithmetic, every expression which overflows is given a reasonable extremum value, either the maximum or the minimum value the expression can hold. For instance, in the above example, if x1==x2==INT_MAX, the expression overflows and saturates, and y is given the value INT_MAX. Saturation is controlled by setting the saturation bit, ST1_SATD, by using these instructions:

```
   BSET ST1_SATD
   BCLR ST1_SATD
```

The compiler must turn this bit on and off to mix saturating and non-saturating arithmetic; however, it minimizes the number of such bit changing instructions by recognizing blocks of instructions with the same behavior. For maximum efficiency, use saturating intrinsic operators for exactly those operations where you need saturated values in case of overflow, and where overflow can occur. Do not use them for loop iteration counters.

The compiler supports associative versions for some of the addition and multiply-and-accumulate intrinsics. These associative intrinsics are prefixed with _a_. The compiler is able to reorder arithmetic computations involving associative intrinsics, which may produce more efficient code.

For example:

```
int x1, x2, x3, y;
y = _a_sadd(x1, _a_sadd(x2, x3)); /* version 1 */
```

can be reordered inside the compiler as:

```
y = _a_sadd(_a_sadd(x1, x2), x3); /* version 2 */
```

However, this reordering may affect the value of the expression if saturation occurs at different points in the new ordering. For instance, if x1==INT_MAX, x2==INT_MAX, and x3==INT_MIN, version 1 of the expression will not saturate, and y will be equal to (INT_MAX-1); however, version 2 will saturate, and y will be equal to -1. A rule of thumb is that if all your data have the same sign, you may safely use associative intrinsics.

Most of the multiplicative intrinsic operators operate in fractional-mode arithmetic. Conceptually, the operands are Q15 fixed-point values, and the result is a Q31 value. Operationally, this means that the result of the normal multiplication is left shifted by one to normalize to a Q31 value. This mode is controlled by the fractional mode bit, ST1_FRCT.

The intrinsics in Table 6-14 are special in that they accept pointers and references to values; the arguments are passed by reference rather than by value. These values must be modifiable values (for example, variables but not constants, nor arithmetic expressions). These intrinsics do not return a value; they create results by modifying the values that were passed by reference. These intrinsics depend on the C++ reference syntax, but are still available in C code with the C++ semantics.

No declaration of the intrinsic functions is necessary, but declarations are provided in the header file, c55x.h, included with the compiler.

Many of the intrinsic operators are useful for implementing basic DSP functions described in the Global System for Mobile Communications (GSM) standard of the European Telecommunications Standards Institute (ETSI). These functions have been implemented in the header file, gsm.h, included with the compiler. Additional support for ETSI GSM functions is described in Section 6.5.5.2.

6.5.5.1 Descriptions of C55x Intrinsics

Table 6-9 through Table 6-15 list all of the intrinsic operators in the TMS320C55x C/C++ compiler. A *function* prototype is presented for each intrinsic that shows the expected type for each parameter. If the argument type does not match the parameter, type conversions are performed on the argument. Where argument order matters, the order of the intrinsic's input arguments matches that of the underlying hardware instruction. The resulting assembly language mnemonic is given for each instruction; for some instructions, such as MPY, an alternate instruction such as SQR (which is a specialized MPY) may be generated if it is more efficient. A brief description is provided for each intrinsic. For a precise definition of the underlying instruction, see the *TMS320C55x DSP Mnemonic Instruction Set Reference Guide* and *TMS320C55x DSP Algebraic Instruction Set Reference Guide*.

Table 6-8. C55x Circular Addressing Intrinsics

Compiler Intrinsic	Description
int _circ_incr(int index, int incr, unsigned int size)	Returns the circular increment of index+incr relative to size when these preconditions are met: 0 <= index < size and incr <= size.

Table 6-9. C55x C/C++ Compiler Intrinsics (Addition, Subtraction, Negation, Absolute Value)

Compiler Intrinsic		Assembly Instruction	Description
int	_sadd(int src1, int src2)	ADD	Returns the saturated sum of its operands.
int	_a_sadd(int src1, int src2)		
long	_lsadd(long src1, long src2)		
long	_a_lsadd(long src1, long src2)		
long long	_llsadd(long long src1, long long src2)		
long long	_a_llsadd(long long src1, long long src2)		
int	_ssub(int src1, int src2)	SUB	Returns the saturated value of the expression (src1 − src2).
long	_lssub(long src1, long src2)		
long long	_llssub(long long src1, long long src2)		
int	_sneg(int src)	NEG	Returns the saturated value of the expression (0 − src).
long	_lsneg(long src)		
long long	_llsneg(long long src)		
int	_abss(int src)	ABS	Returns the saturated absolute value of its operands.
long	_labss(long src)		
long long	_llabss(long src)		

Table 6-10. TMS320C55x C/C++ Compiler Intrinsics (Multiplication)

Compiler Intrinsic		Assembly Instruction	Description
int	_smpy(int src1, int src2)	MPY	Returns the saturated fractional-mode product of its operands.
long	_lsmpy(int src1, int src2)		
long	_lsmpyu(unsigned src1, unsigned src2)		
long	_lsmpysu(int src1, unsigned src2)		
long long	_llsmpy(int sr1, int src2)		
long long	_llsmpyu(unsigned src1, unsigned src2)		
long long	_llsmpysu(int sr1, unsigned src2)		
long	_lmpy(int src1, int src2)	MPY	Returns the unsaturated integer-mode (non-fractional-mode) product of its operands.
long	_lmpyu(unsigned src1, unsigned src2)		
long	_lmpysu(int src1, unsigned src2)		
long long	_llmpy(int src1, int src2)		
long	_lsmpyi(int src1, int src2)	MPY	Returns the saturated integer-mode product of its operands.
long	_lsmpyui(unsigned src1, unsigned src2)		
long	_lsmpysui(int src1, unsigned src2)		
long long	_llsmpyi(int src1, int src2)		
long long	_llsmpyui(unsigned src1, unsigned src2)		
long long	_llsmpysui(int src1, unsigned src2)		
long	_lsmpyr(int src1, int src2)	MPYR	Returns the saturated fractional-mode product of its operands, rounded as if the intrinsic _sround were used.
long	_smac(long src1, int src2, int src3)	MAC	Returns the saturated sum of src1 and the fractional-mode product of src2 and src3.
long	_a_smac(long src1, int src2, int src3)		
long	_smacsu(long src1, int src2, unsigned src3)		
long long	_llsmac(long long src1, int src2, int src3)		
long long	_llsmacu(long long src1, unsigned src2, unsigned src3)		
long long	_llsmacsu(long long src1, int src2, unsigned src3)		
long	_smaci(long src1, int src2, int src3)	MAC	Returns the saturated sum of src1 and the integer-mode product of src2 and src3.
long	_smacsui(long src1, int src2, unsigned src3)		
long long	_llsmaci(long long src1, int src2, int src3)		
long long	_llsmacui(long long src1, unsigned src2, unsigned src3)		
long long	_llsmacsui(long long src1, int src2, unsigned src3)		
long	_smacr(long src1, int src2, int src3)	MACR	Returns the saturated sum of src1 and the fractional-mode product of src2 and src3. The sum is rounded as if the intrinsic _sround were used.
long	_a_smacr(long src1, int src2, int src3)		

Table 6-10. TMS320C55x C/C++ Compiler Intrinsics (Multiplication) (continued)

Compiler Intrinsic		Assembly Instruction	Description
long long long long long long long long long	_smas(long src1, int src2, int src3) _a_smas(long src1, int src2, int src3) _smassu(long src1, int src2, unsigned src3) _llsmas(long long src1, int src2, int src3) _llsmasu(long long src1, unsigned src2, unsigned src3) _llsmassu(long long src1, int src2, unsigned src3)	MAS	Returns the saturated difference of src1 and the fractional-mode product of src2 and src3.
long long long long long long long long	_smasi(long src1, int src2, int src3) _smassui(long src1, int src2, unsigned src3) _llsmasi(long long src1, int src2, int src3) _llsmasui(long long src1, unsigned src2, unsigned src3) _llsmassui(long long src1, int src2, unsigned src3)	MAS	Returns the saturated difference of src1 and the integer-mode product of src2 and src3.

long
long
	_smasr(long src1, int src2, int src3)
	a_smasr(long src1, int src2, int src3)
		MASR	Returns the saturated difference of src1 and the fractional-mode product of src2 and src3. The sum is rounded as if the intrinsic _sround were used.

Table 6-11. TMS320C55x C/C++ Compiler Intrinsics (Shifting)

Compiler Intrinsic		Assembly Instruction	Description

int
long
long long
	_sshl(int src1, int src2)
	_lsshl(long src1, int src2)
	_llsshl(long long, int)
		SFTS	Returns the saturated value of the expression (src1<<src2). If src2 is negative, a right shift is performed instead.

int
long
	_shrs(int src1, int src2)
	_lshrs(long src1, int src2)
		SFTS	Returns the saturated value of the expression (src1>>src2). If src2 is negative, a left shift is performed instead.

int
long
long long
	_shl(int src1, int src2)
	_lshl(long src1, int src2)
	_llshl(long long src1, int src2)
		SFTS	Returns the expression (src1<<src2). If src2 is negative, a right shift is performed instead. No saturation is performed.

Table 6-12. TMS320C55x C/C++ Compiler Intrinsics (Shifting and Storing)

Compiler Intrinsic		Description
void	_llsshlstore(long long src, int cnt, int *dst)	Stores bits 16-31 of the result of a saturating (based on bit 31) shift of src by cnt into dst using: MOV HI(saturate(src << cnt)) , dst
void void	_llsshlstorer(long long, int, int*) _llsshlstorern(long long, int, int*)	Stores bits 16-31 of the rounded result of a saturating (based on bit 31) shift of src by cnt into dst using: MOV rnd(HI(saturate(src << cnt))), dst _llsshlstorer rounds by adding 2^{15} using saturating arithmetic (biased round to positive infinity). _llsshlstorern rounds to nearest multiple of 2^{16} using saturating arithmetic. Ties are broken by rounding to even.
void void	_llshlstorer(long long, int, int*) _llshlstorern(long long, int, int*)	Stores bits 16-31 of the rounded result of shifting src by cnt using: MOV rnd(HI(src << cnt)), dst _llshlstorer rounds by adding 2^{15} using unsaturating arithmetic (biased round to positive infinity). _llshlstorern rounds to nearest multiple of 2^{16} using unsaturating arithmetic. Ties are broken by rounding to even.

Table 6-13. TMS320C55x C/C++ Compiler Intrinsics (Rounding, Saturation, Bitcount, Extremum)

Compiler Intrinsic		Assembly Instruction	Description

long
long long
	_round(long src)
	_llround(long long src)
		ROUND	Uses unsaturating arithmetic (biased round to positive infinity) and clears the lower 16 bits. The upper 16 bits of the Q31 result can be treated as a Q15 value.

long
long long
	_sround(long src)
	_llsround(long long src)
		ROUND	Returns the value src rounded by adding 2^15 using saturating arithmetic (biased round to positive infinity) and clearing the lower 16 bits. The upper 16 bits of the Q31 result can be treated as a Q15 value.

long
long long
	_roundn(long src)
	_llroundn(long long src)
		ROUND	Returns the value src rounded to the nearest multiple of 2^16 using unsaturating arithmetic and clearing the lower 16 bits. Ties are broken by rounding to even. The upper 16 bits of the Q31 result can be treated as a Q15 value.

long
long long
	_sroundn(long src)
	_llsroundn(long long src)
		ROUND	Returns the value src rounded to the nearest multiple of 2^16 using saturating arithmetic and clearing the lower 16 bits. Ties are broken by rounding to even. The upper 16 bits of the Q31 result can be treated as a Q15 value.

int
int
int
	_norm(int src)
	_lnorm(long src)
	_llnorm(long long src)
		EXP	Returns the left shift count needed to normalize src to a 32-bit long value. This count may be negative.

long	_lsat(long long src)	SAT	Returns src saturated to a 32-bit long value. If src was already within the range allowed by long, the value does not change; otherwise, the value returned is either LONG_MIN or LONG_MAX.
int	_count(unsigned long long src1, unsigned long long src2)	BCNT	Returns the number of bits set in the expression (src1 & src2).

int
long
long long
	_max(int src1, int src2)
	_lmax(long src1, long src2)
	_llmax(long long src1, long long src2)
		MAX	Returns the maximum of src1 and src2.

int
long
long long
	_min(int src1, int src2)
	_lmin(long src1, long src2)
	_llmin(long long src1, long long src2)
		MIN	Returns the minimum of src1 and src2.

Table 6-14. Compiler Intrinsics (Arithmetic With Side Effects)

Compiler Intrinsic	Assembly Instruction	Description

void _firs(int *, int *, int *, int&, long&)
void _firsn(int *, int *, int *, int&, long&)
	FIRSADD
	FIRSSUB
		Performs the corresponding instruction as follows:
		int *p1, *p2, *p3, srcdst1;
		long srcdst2;
		...
		_firs(p1, p2, p3, srcdst1, srcdst2);
		_firsn(p1, p2, p3, srcdst1, srcdst2);
		Which becomes (respectively):
		FIRSADD *p1, *p2, *p3, srcdst1, srcdst2
		FIRSSUB *p1, *p2, *p3, srcdst1, srcdst2
		Mode bits SATD, FRCT, and M40 are 0.

void _lms(int *, int *, int&, long&)	LMS	Performs the LMS instruction as follows: Where *type* is long or long long int *p1, *p2, srcdst1; *type* srcdst2; ... _lms (p1, p2, srcdst1, srcdst2); Which becomes: LMS *p1, *p2, srcdst1, srcdst2 For _llslms and _llslmsi saturation is enabled. For _llslms fractional mode is enabled

Table 6-14. Compiler Intrinsics (Arithmetic With Side Effects) (continued)

Compiler Intrinsic	Assembly Instruction	Description

void _abdst(int *, int *, int&, long&)
void _sqdst(int *, int *, int&, long&)
	ABDST
	SQDST
		Performs the corresponding instruction as follows:
		int *p1, *p2, srcdst1;
		long srcdst2;
		...
		_abdst(p1, p2, srcdst1, dst);
		_sqdst(p1, p2, srcdst1, dst);
		Which becomes (respectively):
		ABDST *p1, *p2, srcdst1, srcdst2
		SQDST *p1, *p2, srcdst1, srcdst2
		Mode bits SATD, FRCT, and M40 are 0.

int _exp_mant(long, long&) int _llexp_mant(long long, long long&)	MANT:: NEXP	Performs the MANT::NEXP instruction pair, as follows: int src, dst2; long dst1; ... dst2 = _exp_mant(src, dst1); Which becomes: MANT src, dst1 :: NEXP src, dst2

void _max_diff_dbl(long, long, long&, long&, unsigned &)
void _min_diff_dbl(long, long, long&, long&, unsigned &)
void _smax_diff_dbl(long, long, long&, long&, unsigned&)
void _smin_diff_dbl(long, long, long&, long&, unsigned&)
void _llmax_diff_dbl(long long, long long, long long&, long long&, unsigned&)
void _llmin_diff_dbl(long long, long long, long long&, long long&, unsigned&)
void _llsmax_diff_dbl(long long, long long, long long&, long long&, unsigned&)
void _llsmin_diff_dbl(long long, long long, long long&, long long&, unsigned&)
	DMAXDIFF
	DMINDIFF
		Performs the corresponding instruction, as follows:
		Where *type* is long or long long
		type src1, src2, dst1, dst2;
		int dst3;
		...
		_max_diff_dbl(src1, src2, dst1, dst2, dst3);
		_min_diff_dbl(src1, src2, dst1, dst2, dst3);
		Which becomes (respectively):
		DMAXDIFF src1, src2, dst1, dst2, dst3
		DMINDIFF src1, src2, dst1, dst2, dst3
		The smax and smin forms are performed with saturation enabled.

Table 6-15. C55x C/C++ Compiler Intrinsics (Non-Arithmetic)

Compiler Intrinsic	Assembly Instruction	Description
ong long _dtol(double)		Reinterpret double as long (when long is 40 bits).
long long _dtoll(double)		Reinterpret double as long long.
void _enable_interrupts(void) unsigned int _disable_interrupts(void)	BCLR ST1_INTM BSET ST1_INTM	Enables or disables interrupts and ensure enough cycles are consumed that the change takes effect before anything else happens.
void _restore_interrupts(unsigned int)		Restores interrupts to state indicated by value returned from _disable_interrupts .

6.5.5.2 Intrinsics and ETSI Functions

The functions in Table 6-16 provide additional support for ETSI GSM functions. Functions L_add_c, L_sub_c, and L_sat map to GSM inline macros. The other functions in the table are run-time functions. Additional details about these functions can be found in various ETSI documents. In particular, see Chapter 13 (BASOP: ITU-T Basic Operators) of *ITU-T Software Tool Library 2005 User's Manual* found at http://www.itu.int/rec/T-REC-G.191-200508-I/en.

Table 6-16. ETSI Support Functions

Compiler Intrinsic	Description
long L_add_c(long src1, long src2)	Adds src1, src2, and Carry bit. This function does not map to a single assembly instruction, but to an inline function.
long L_sub_c(long src1, long src2)	Subtracts src2 and logical inverse of sign bit from src1. This function does not map to a single assembly instruction, but to an inline function.

Table 6-16. ETSI Support Functions (continued)

Compiler Intrinsic	Description
long L_sat(long src1)	Saturates any result after L_add_c or L_sub_c if Overflow is set.
int crshft_r(int x, int y)	Shifts x right by y, rounding the result with saturation.
long L_crshft_r(long x, int y)	Shifts x right by y, rounding the result with saturation.
int divs(int x, int y)	Divides x by y with saturation.

Example 6-11. Intrinsics Header File, gsm.h

```
#ifndef _GSMHDR
#define _GSMHDR
#include <linkage.h>
#define MAX_16 0x7fff
#define MIN_16 -32768
#define MAX_32 0x7fffffff
#define MIN_32 0x80000000
extern int Overflow;
extern int Carry;
#define L_add(a,b) (_lsadd((a),(b)))
#define L_sub(a,b) (_lssub((a),(b)))
#define L_negate(a) (_lsneg(a))
#define L_deposit_h(a) ((long)a<<16)
#define L_deposit_l(a) ((long)a)
#define L_abs(a) (_labss((a)))
#define L_mult(a,b) (_lsmpy((a),(b)))
#define L_mac(a,b,c) (_smac((a),(b),(c)))
#define L_macNs(a,b,c) (L_add_c((a),L_mult((b),(c))))
#define L_msu(a,b,c) (_smas((a),(b),(c)))
#define L_msuNs(a,b,c) (L_sub_c((a),L_mult((b),(c))))
#define L_shl(a,b) _lsshl((a),(b))
#define L_shr(a,b) _lshrs((a),(b))
#define L_shr_r(a,b) (L_crshft_r((a),(b)))
#define L_shift_r(a,b) (L_shr_r((a),-(b)))
#define abs_s(a) (_abss((a)))
#define add(a,b) (_sadd((a),(b)))
#define sub(a,b) (_ssub((a),(b)))
#define extract_h(a) ((unsigned)((a)>>16))
#define extract_l(a) ((int)a)
#define round(a) (short)(_sround(a)>>16)
#define mac_r(a,b,c) (short)(_smacr((a),(b),(c))>>16)
#define msu_r(a,b,c) (short)(_smasr((a),(b),(c))>>16)
#define mult_r(a,b) (short)(_lsmpyr((a),(b))>>16)
#define mult(a,b) (_smpy((a),(b)))
#define norm_l(a) (_lnorm(a))
#define norm_s(a) (_norm(a))
#define negate(a) (_sneg(a))
#define shl(a,b) _sshl((a),(b))
#define shr(a,b) _shrs((a),(b))
#define shr_r(a,b) (crshft_r((a),(b)))
#define shift_r(a,b) (shr_r(a,-(b)))
#define div_s(a,b) (divs(a,b))
#ifdef __cplusplus
extern "C"
{
#endif /* __cplusplus */

int          crshft_r(int x, int y);
long         L_crshft_r(long x, int y);
int          divs(int x, int y);
_IDECL long  L_add_c(long, long);
_IDECL long  L_sub_c(long, long);
_IDECL long  _sat(long);
```

Example 6-11. Intrinsics Header File, gsm.h (continued)

```
#ifdef _INLINE
static inline long L_add_c (long L_var1, long L_var2)
{
   unsigned long uv1 = L_var1;
   unsigned long uv2 = L_var2;
   int cin = Carry;
   unsigned long result = uv1 + uv2 + cin;

   Carry = ((~result & (uv1 | uv2)) | (uv1 & uv2)) >> 31;
   Overflow = ((~(uv1 ^ uv2)) & (uv1 ^ result)) >> 31;

   if (cin && result == 0x80000000) Overflow = 1;
   return (long)result;
}


static inline long L_sub_c (long L_var1, long L_var2)
{
   unsigned long uv1 = L_var1;
   unsigned long uv2 = L_var2;
   int cin = Carry;
   unsigned long result = uv1 + ~uv2 + cin;

   Carry = ((~result & (uv1 | ~uv2)) | (uv1 & ~uv2)) >> 31;
   Overflow = ((uv1 ^ uv2) & (uv1 ^ result)) >> 31;

   if (!cin && result == 0x7fffffff) Overflow = 1;
   return (long)result;
}

static inline long L_sat (long L_var1)
{
   int cin = Carry;
   return !Overflow ? L_var1 : (Carry = Overflow = 0, 0x7fffffff+cin);
}
#endif /* !_INLINE */

#ifdef __cplusplus
} /* extern "C" */
#endif /* __cplusplus */
#endif /* !_GSMHDR */
```

Definitions of Function round

NOTE: Both the header files math.h and gsm.h contain definitions for the function round. (The former introduces the round function defined by the C99 standard.) Thus, a conflict will occur if both of these header files are included in the same file. Including the two files in the same source file results in a warning of an incompatible redefinition of the macro round.

The diagnostic is only a warning and the last encountered definition of round will be used. The warning may be turned into an error using the -pdse48 option or "#pragma DIAG_ERROR 48".

6.6 Interrupt Handling

As long as you follow the guidelines in this section, you can interrupt and return to C/C++ code without disrupting the C/C++ environment. When the C/C++ environment is initialized, the startup routine does not enable or disable interrupts. If the system is initialized by way of a hardware reset, interrupts are disabled. If your system uses interrupts, you must handle any required enabling or masking of interrupts. Such operations have no effect on the C/C++ environment and are easily incorporated with asm statements or calling an assembly language function.

6.6.1 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.

6.6.2 Saving Context on Interrupt Entry

All registers that the interrupt routine uses, including the registers that are normally parent-saved, must be preserved. If the interrupt routine calls other functions, there is the additional requirement that *all* registers, whether used or not, that are normally parent-saved must be preserved. This is because the called function assumes these registers are available for use, but the caller of the interrupt function, namely the interrupted function, does not have a chance to save these registers.

6.6.3 Using C/C++ Interrupt Routines

Interrupts can be handled directly with C/C++ functions by using the interrupt keyword. For example:

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

Adding the interrupt keyword defines an interrupt routine. When the compiler encounters one of these routines, it generates code that allows the function to be activated from an interrupt trap, performing the necessary stack alignment and context save. This method provides more functionality than the standard C/C++ signal mechanism, which is not implemented in the C55x library. This does not prevent implementation of the signal function, but it does allow these functions to be written entirely in C/C++.

6.6.4 Intrinsics for Interrupts

These intrinsics support enabling and disabling interrupts from C. They are defined in the file c55x.h and are the following:

```
void _enable_interrupts(void);
unsigned int _disable_interrupts(void);
void _restore_interrupts(unsigned int);
```

These functions compile into instructions that clear (for enable) or set (for disable) the INTM bit in the ST1 status register.

When the value returned by a call to _disable_interrupts is passed as the argument to _restore_interrupts, the result is that interrupts are re-enabled only if they were enabled prior to the call to _disable_interrupts.

The value returned by _disable_interrupts is not defined other than for the use described above. The behavior of _restore_interrupts is undefined unless the value passed as the argument was previously returned by a call to _disable_interrupts.

The value returned by _disable_interrupts reflects the state of interrupts at the point of the call, but not necessarily at the point when the interrupts are actually disabled. In normal use this should not be a problem. It is due to the fact that the implementation of _disable_interrupts is not atomic with respect to reading the existing value of the global interrupts masked status bit (INTM) and setting it. An interrupt service routine that exits with interrupts disabled can cause the value returned by _disable_interrupts to indicate interrupts enabled even though at the point of disabling they are already disabled. Other than this somewhat unusual case, the value reflects the state at the point of disabling.

On some versions of the hardware there is a latency between setting the INTM bit and when interrupts are actually disabled. The compiler manages this latency by scheduling the instruction that sets INTM an appropriate number of cycles before the point where _disable_interrupts() appears in the source code. (The actual point at which interrupts are disabled is indicated by an interrupts disabled comment in the compiler's assembly output.)

Using these intrinsics not only allows you to enable and disable the interrupts with C code but guarantees that any necessary timing issues are handled automatically.

6.7 Extended Addressing of Data in P2 Reserved Mode

The run-time library includes functions to support reading and writing of data in the full TMS320C55x address space. These functions may be accessed via extaddr.h, shown in Example 6-12. Extended memory addresses are represented by values of the integer type FARPTR (unsigned long). An extended addressing code example is shown in Example 6-14.

When using P2 Restricted Mode hardware all C-accessible data objects must reside on the 64K-word data page zero.

Run-time functions have been added to support copying data to and from remote data pages and page zero. Full, 23-bit addresses passed to these functions are represented as unsigned long integer values.

The .cinit sections and .switch sections can be placed in extended memory. However, .const sections cannot be placed in extended memory. For more information, see Section 6.8.

Example 6-12. The File extaddr.h

```
/**********************************************************************/
/* extaddr.h                                                          */
/**********************************************************************/

/**********************************************************************/
/* Type of extended memory data address values                        */
/**********************************************************************/

   typedef unsigned long FARPTR;

/**********************************************************************/
/* Prototypes for Extended Memory Data Support Functions              */
/*                                                                    */
/* far_peek       Read an int from extended memory address            */
/* far_peek_l     Read an unsigned long from extended memory address  */
/* far_poke       Write an int to extended memory address             */
/* far_poke_l     Write an unsigned long to extended memory address   */
/* far_memcpy     Block copy between extended memory addresses        */
/* far_near_memcpy  Block copy from extended memory address to page 0 */
/* near_far_memcpy  Block copy from page 0 to extended memory address */
/**********************************************************************/

int far_peek(FARPTR);
unsigned long far_peek_l(FARPTR);
void far_poke(FARPTR, int);
void far_poke_l(FARPTR, unsigned long);

void far_memcpy(FARPTR, FARPTR, int);
void far_near_memcpy(void *, FARPTR, int);
void near_far_memcpy(FARPTR, void *, int);
```

6.7.1 Placing Data in Extended Memory

Global and static C variables can be placed in extended memory. Use the DATA_SECTION pragma to place them in a particular named section. Then edit your linker control file to place that named section in extended memory.

You can access such variables only by using the functions described in Section 6.7. Use of such variables in normal C expressions leads to unpredictable results.

Global and static variables placed in extended memory can be initialized in the C source code as usual. The autoinitialization of these variables is supported in the same manner as other global and static variables.

6.7.2 Obtaining the Full Address of a Data Object in Extended Memory

The pragma FAR is used to obtain the full, 23-bit address of a data object declared at file scope. The pragma must immediately precede the definition of the symbol to which it applies as shown below:

```
#pragma FAR(x)
int x;         /* ok */
#pragma FAR(z)
int y;         /* error - z's definition must */
int z;         /*  immediately follow #pragma */
```

This pragma can only be applied to objects declared at file scope. Once this is done, taking the address of the indicated object produces the 23-bit value that is the full address of the object. This value may be cast to FARPTR (that is, unsigned long) and passed to the run-time support functions. For a structured object taking the address of a field also produces a full address.

6.7.3 Accessing Far Data Objects

Support for accessing far data objects is not general. Declaring an object with the FAR pragma does not cause the compiler to treat it as far for any construct other than taking its address (as shown in Example 6-13). For example, there are no far pointer types and far objects cannot be accessed directly (as in i = ary[10]). The compiler does not diagnose these transgressions.

Example 6-13. Idiom for Accessing a Far Object

```
#include  <extaddr.h>

#pragma   DATA_SECTION(var, ".far_data")
#pragma   FAR(var)
int var;

#pragma   DATA_SECTION(ary, ".far_data")
#pragma   FAR(ary)
int       ary[100];
   ...

void      func(void)
{
   /* Save pointer to ary */
   FARPTR aptr = (FARPTR) &ary;
   ...

   /* Load page zero from extended memory */
   i = far_peek((FARPTR) &var);
   far_near_copy(&data, aptr, 100);
   ...
```

6.8 The .const Sections in Extended Memory

Unlike .cinit and .switch sections, .const sections cannot be placed in extended memory. However, the -mc shell option allows constants normally placed in a .const section to be treated as read-only, initialized static variables. This allows the constant values to be loaded into extended memory while the space used to hold the values at run time is still in page 0.

The space used to hold the constant is allocated in a .bss section. An autoinitialization record used to initialize the constant is allocated in a .cinit section. Autoinitialization places the constant value into the location in the .bss section in the same way as it does for the initialization of the global and static variables.

A constant explicitly placed in a named section via the DATA_SECTION pragma is not affected by the -mc option. It will be placed in the named section, not in a .cinit record for initialization.

Example 6-14. Extended Addressing of Data

```
#include <extaddr.h>

/**************************************************/
/* Variables to be placed in extended memory.     */
/**************************************************/
#pragma   DATA_SECTION(ival, ".far_data")
#pragma   FAR(ival)
int ival;

#pragma   DATA_SECTION(lval, ".far_data")
#pragma   FAR(lval)
unsigned  long lval;

#pragma   DATA_SECTION(a, ".far_data")
```

Example 6-14. Extended Addressing of Data (continued)

```
#pragma   FAR(a)
int       a[10] = {0,1,2,3,4,5,6,7,8,9};

#pragma   DATA_SECTION(b, ".far_data")
#pragma   FAR(b)
int       b[10];

#pragma   DATA_SECTION(c, ".far_data")
#pragma   FAR(c)
char      c[12] = {"test string"};

/**************************************************/
/* Variable to be placed in memory page 0.        */
/**************************************************/
char   d[12];


void   main(void)
{
int ilocal;

/* Get extended addresses of variables */
FARPTR iptr  =  (FARPTR) &ival;
FARPTR lptr  =  (FARPTR) &lval;

/**************************************************/
/* Read and write variables in extended memory    */
/**************************************************/

/* ival = 100 */
far_poke(iptr, 100);

/* ival += 10 */
ilocal = far_peek(iptr) + 10;
far_poke(iptr, ilocal);

/* lval = 0x7ffffffe */
far_poke_l(lptr, 0x7ffffffe);

/* lval += 1 */
far_poke_l(lptr, far_peek_l(lptr) + 1);

/**************************************************/
/* Copy string from extended data memory to an    */
/* address in Page 0.                             */
/**************************************************/
far_near_memcpy((void *) d, (FARPTR) &c, 12);

/**************************************************/
/* Copy an array of integers from one extended    */
/* address to another extended address.           */
/**************************************************/
far_memcpy((FARPTR) &b, (FARPTR) &a, 10);
}
```

6.9 System Initialization

Before you can run a C/C++ program, you must create the C/C++ run-time environment. The C/C++ boot routine performs this task using a function called c_int00 (or _c_int00). The run-time-support source library, rts.src, contains the source to this routine in a module named boot.c (or boot.asm).

To begin running the system, the c_int00 function can be 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. Sets up the stack and the secondary system stack.
2. Defines a section called .stack for the system stack and sets up the initial stack pointers
3. Initializes global variables by copying the data from the initialization tables to the storage allocated for the variables in the .bss section. If you are initializing variables at load time (--ram_model option), a loader performs this step before the program runs (it is not performed by the boot routine). For more information, see Section 6.9.1.
4. Executes the global constructors found in the global constructors table. For more information, see Section 6.9.1.4.
5. Calls the function main to run the C/C++ program

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the operations listed above to correctly initialize the C/C++ environment.

For additional information on the boot routine, see Section 4.3.2.

6.9.1 Automatic Initialization of Variables

Some global variables must have initial values assigned to them before a C/C++ program starts running. The process of retrieving these variables' data and initializing the variables with the data is called autoinitialization.

The compiler builds tables in a special section called .cinit that contains data for initializing global and static variables. Each compiled module contains these initialization tables. The linker combines them into a single table (a single .cinit section). The boot routine or a loader uses this table to initialize all the system variables.

Initializing Variables

NOTE: In ANSI/ISO C, global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ compiler does not perform any preinitialization of uninitialized variables. Explicitly initialize any variable that must have an initial value of 0.

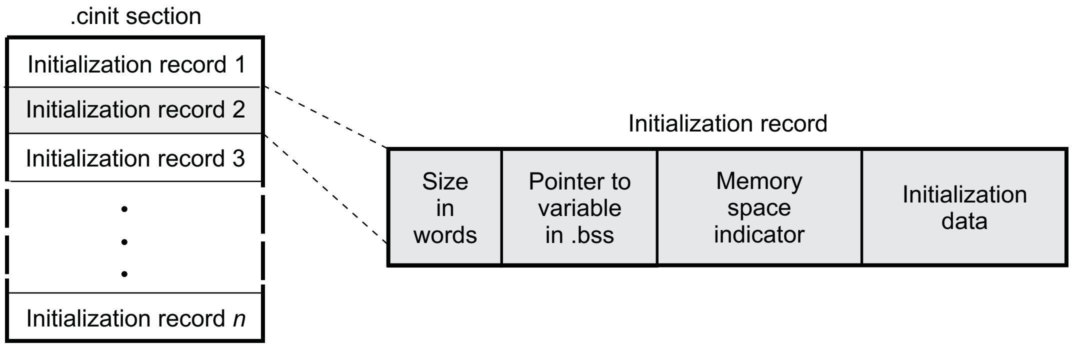
The easiest method is to set a fill value of zero in the linker control map for the .bss section. (You cannot use this method with code that is burned into ROM).

Global variables are either autoinitialized at run time or at load time; see Section 6.9.1.2 and Section 6.9.1.3. Also see Section 5.12.

6.9.1.1 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 6-3 shows the format of the .cinit section and the initialization records.

Figure 6-3. Format of Initialization Records in the .cinit Section



The fields of an initialization record contain the following information:

- The first field (word 0) contains the size in words of the initialization data.

  Bits 14 and 15 are reserved and must be set to zero. An initialization record can contain up to 2^{13} -1 words of initialization data, plus the size of the header.
- The second field contains the starting address of the area in the .bss section where the initialization data must be copied. This field is 24 bits to accommodate an address greater than 16 bits.
- The third field contains 8 bits of flags. Bit 0 is a flag for the memory space indicator (I/O or data). The other bits are reserved and set to zero.
- The fourth field (words 3 through n) contains the data that is copied to initialize the variable.

The .cinit section contains an initialization record for each variable that is initialized. Example 6-15 shows initialized variables defined in C/C++. Example 6-16 shows the corresponding initialization table.

Example 6-15. Initialized Variables Defined in C

```
int    i = 3;
long   x = 4;
float  f = 1.0;
char   s[] = "abcd";
long   a[5] = { 1, 2, 3, 4, 5 };
```

Example 6-16. Initialized Information for Variables Defined in Example 6-15

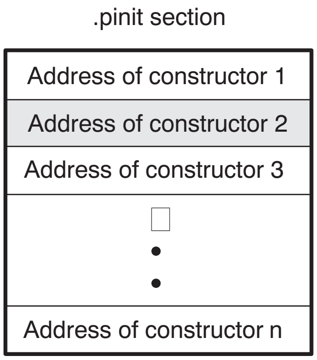
```
      .sect ".cinit"          ; Initialization section
* Initialization record for variable i
      .field    1,16          ; length of data (1 word)
      .field    _i+0,24       ; address in .bss
      .field    0,8           ; signifies data memory
      .field    3,16          ; int is 16 bits
* Initialization record for variable x
      .field    2,16          ; length of data (2 words)
      .field    _l+0,24       ; address in .bss
      .field    0,8           ; data memory
      .field    4,32          ; long is 32 bits
* Initialization record for variable f
      .field    2,16          ; length of data (2 words)
      .field    _f+0,24       ; address in .bss
      .field    0,8           ; data memory
      .xlong    0x3f800000    ; float is 32 bits
* Initialization record for variable s
      .field    IR_1,16       ; length of data
      .field    _s+0,24       ; address in .bss
      .field    0,8           ; data memory
      .field    97,16         ; a
      .field    98,16         ; b
      .field    99,16         ; c
      .field    100,16        ; d
      .field    0,16          ; end of string
IR_1 .set 5                   ; symbolic value gives

                              ; count of elements
* Initialization record for variable a
      .field    IR_2,16       ; length of data
      .field    _a+0,24       ; address in .bss
      .field    0,8           ; data memory
      .field    1,32          ; beginning of array
      .field    2,32
      .field    3,32
      .field    4,32
      .field    5,32          ; end of array
IR_2 .set 10                  ; size of array
```

The .cinit section must contain only initialization tables in this format. When interfacing assembly language modules, do not use the .cinit section for any other purpose.

The table in the .pinit section simply consists of a list of addresses of constructors to be called (see Figure 6-4). The constructors appear in the table after the .cinit initialization.

Figure 6-4. Format of Initialization Records in the .pinit Section



.pinit section

Address of constructor 1

Address of constructor 2

Address of constructor 3

□

•

•

Address of constructor n

When you use the --rom_model or --ram_model option, the linker combines the .cinit sections from all the C modules and appends a null word to the end of the composite .cinit section. This terminating record appears as a record with a size field of 0 and marks the end of the initialization tables.

Likewise, the --rom_model or --ram_model link option causes the linker to combine all of the .pinit sections from all C/C++ modules and append a null word to the end of the composite .pinit section. The boot routine knows the end of the global constructor table when it encounters a null constructor address.

The const-qualified variables are initialized differently; see Section 5.5.1.

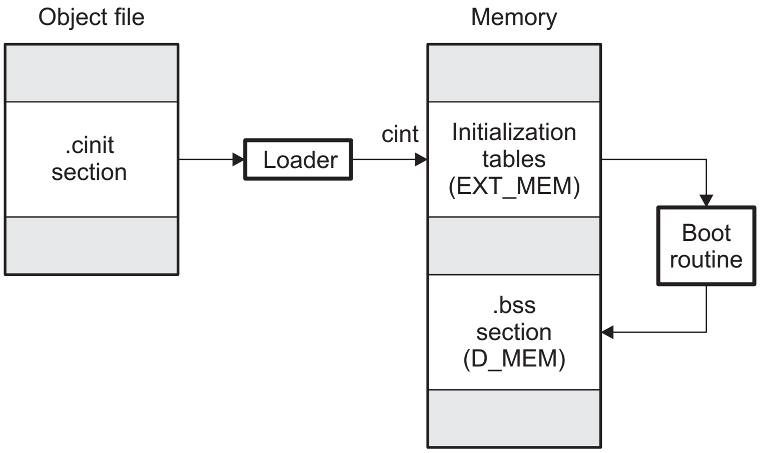
6.9.1.2 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the --rom_model option.

Using this method, the .cinit section is loaded into memory along with all the other initialized sections, and global variables are initialized at run time. The linker defines a special symbol called cinit that points to the beginning of the initialization tables in memory. When the program begins running, the C/C++ boot routine copies data from the tables (pointed to by .cinit) into the specified variables in the .bss section. This allows initialization data to be stored in ROM and copied to RAM each time the program starts.

Figure 6-5 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

Figure 6-5. Autoinitialization at Run Time



6.9.1.3 Initialization of Variables at Load Time

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the --ram_model option.

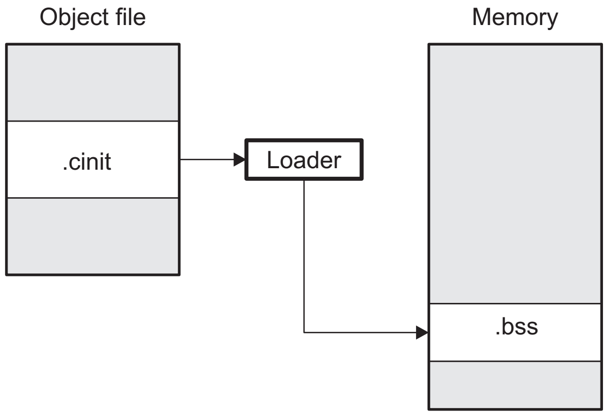
When you use the --ram_model link option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to -1 (normally, cinit points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run-time initialization is performed at boot time.

A loader (which is not part of the compiler package) must be able to perform the following tasks to use initialization at load time:

- Detect the presence of the .cinit section in the object file
- Determine that STYP_COPY is set in the .cinit section header, so that it knows not to copy the .cinit section into memory
- Understand the format of the initialization tables

Figure 6-6 illustrates the initialization of variables at load time.

Figure 6-6. Initialization at Load Time



Regardless of the use of the --rom_model or --ram_model options, the .pinit section is always loaded and processed at run time.

6.9.1.4 Global Constructors

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.

Chapter 7

SPRU281G–December 2011

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 complete ISO standard library except for those facilities that handle locale issues (properties that depend on local language, nationality, or culture). Using the ANSI/ISO standard library ensures a consistent set of functions that provide for greater portability.

In addition to the ANSI/ISO-specified functions, the run-time-support library includes routines that give you processor-specific commands and direct C language I/O requests. These are detailed inSection 7.1 and Section 7.2.

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 7.4 .

Topic **Page**

7.1 C and C++ Run-Time Support Libraries

TMS320C55x compiler releases include pre-built run-time libraries that provide all the standard capabilities. Separate libraries are provided for C++ exception support. See Section 7.4 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
- Functions and macros that allow C/C++ to access specific instructions

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 5.1.

The C++ library included with the compiler is licensed from Dinkumware, Ltd. The Dinkumware C++ library is a fully conforming, industry-leading implementation of the standard C++ library.

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
- Dinkumware's online reference at http://dinkumware.com/manuals

7.1.1 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 *TMS320C55x 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.

7.1.2 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 C55X_C_DIR environment variable to the include directory where the tools are installed.

7.1.3 Modifying a Library Function

You can inspect or modify library functions by unzipping the source file (rtssrc.zip), changing the specific function file, and rebuilding the library. When extracted (with any standard unzip tool on windows, linux, or unix), this zip file recreates the run-time source tree for the run-time library.

The source for the libraries is included in the rtssrc.zip file. See Section 7.4 for details on rebuilding.

You can also build a new library this way, rather than rebuilding into rts55.lib. See Section 7.4.

7.1.4 Minimal Support for Internationalization

The library now 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 multi-byte characters. The type wchar_t is implemented as int. The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h> but does not include all the functions specified in the standard. So-called multi-byte characters are limited to single characters. There are no shift states. The mapping between multi-byte characters and wide characters is simple equivalence; that is, each wide character maps to and from exactly a single multi-byte character having the same value.
- The C library includes the header file <locale.h> but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hard-coded to the behavior of the C locale, and attempting to install a different locale via a call to setlocale() will return NULL.

7.1.5 Allowable Number of Open Files

In the <stdio.h> header file, the value for the macro FOPEN_MAX has been changed from 12 to 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.

7.1.6 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 according to the following naming scheme:

arch[*mem*][*mode*][_eh]

arch	Indicates the architecture:	
	rts55	C55x
memory_model	Indicates the memory model:	
		small memory model by default
	x	large memory model
	h	huge memory model
_eh	Indicates the library has exception handling support	

For example, rts55h_eh.lib is the C55x library that supports the huge memory model and exceptions handling.

7.2 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.

Debugger Required for Default HOST

NOTE: 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*TMS320C55x 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 todal). The macro _NDEVICE controls how many device drivers are installed at one time (the HOST device counts against this total).

7.2.1 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:

```
cl55 main.c --run_linker --heap_size=400 --library=rts55.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.

7.2.2 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 7.2.3 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 7.2.5).
- 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 7.2.3.

The *path* is the filename of the file, including path information and optional device prefix. (See Section 7.2.5.)

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.

7.2.3 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 chose 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 7.2.5 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. It need not be unique across devices. Only the low-level I/O functions will see this device file descriptor; the low-level function open will assign its own unique file descriptor.

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 *** *bu* **, 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 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 by 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 7.2.3.

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.

7.2.4 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 7-1. 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 7-1. Each function must set up and maintain its own data structures as needed. Some function definitions perform no action and should just return.

Example 7-1. 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"); }
```

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.

7.2.5 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 rtssrc.zip

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 7.2.2. The device driver for the HOST that the TMS320C55x 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 7-2 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 7-2 illustrates adding and using a device for C I/O:

Example 7-2. 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);
}
```

7.3 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, BIOS applications have multiple threads which need to modify the same global program state, such as the CIO buffer, so reentrancy is a concern.

Part of the problem of reentrancy remains your responsibility, but the run-time-support environment does provide rudimentary support for multi-threaded reentrancy by providing support for critical sections. This implementation does not protect you from reentrancy issues such as calling run-time-support functions from inside interrupts; this remains your responsibility.

The run-time-support environment provides hooks to install critical section primitives. By default, a single-threaded model is assumed, and the critical section primitives are not employed. In a multi-threaded system such as BIOS, the kernel arranges to install semaphore lock primitive functions in these hooks, which are then called when the run-time-support enters code that needs to be protected by a critical section.

Throughout the run-time-support environment where a global state is accessed, and thus needs to be protected with a critical section, there are calls to the function _lock(). This calls the provided primitive, if installed, and acquires the semaphore before proceeding. Once the critical section is finished, _unlock() is called to release the semaphore.

Usually BIOS is responsible for creating and installing the primitives, so you do not need to take any action. However, this mechanism can be used in multi-threaded applications which do not use the BIOS LCK mechanism.

You should not define the functions _lock() and _unlock() functions directly; instead, the installation functions are called to instruct the run-time-support environment to use these new primitives:

```
void _register_lock  (void (  *lock)());

void _register_unlock(void (*unlock)());
```

The arguments to _register_lock() and _register_unlock() should be functions which take no arguments and return no values, and which implement some sort of global semaphore locking:

```
extern volatile sig_atomic_t *sema = SHARED_SEMAPHORE_LOCATION;
static int sema_depth = 0;
static void my_lock(void)
{
    while (ATOMIC_TEST_AND_SET(sema, MY_UNIQUE_ID) != MY_UNIQUE_ID);
    sema_depth++;
}
static void my_unlock(void)
{
    if (!--sema_depth) ATOMIC_CLEAR(sema);
}
```

The run-time-support nests calls to _lock(), so the primitives must keep track of the nesting level.

7.4 Library-Build Process

When using the C/C++ compiler, you can compile your code under a large number of different configurations and options that are not necessarily compatible with one another. Because it would be infeasible to include all possible run-time-support library variants, compiler releases pre-build only a small number of very commonly-used libraries such as rts55.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

7.4.1 Required Non-Texas Instruments Software

To use the self-contained run-time-support build process to rebuild a library with custom options, the following are required:

- sh (Bourne shell)
- unzip (InfoZIP unzip 5.51 or later, or equivalent)

  You can download the software from http://www.info-zip.org.
- 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 platfortms should expliclty 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.

7.4.2 Using the Library-Build Process

You should normally let the linker automatically rebuild libraries as needed. If necessary, you can run mklib directly to populate libraries. See Section 7.4.2.2 for situations when you might want to do this.

7.4.2.1 Automatic Standard Library Rebuilding by the Linker

The linker looks for run-time-support libraries primarily through the C55X_C_DIR environment variable. Typically, one of the pathnames in C55X_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 C55X_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 is explicitly named (e.g. rts55.lib), run-time support looks for that library exactly; otherwise, it uses the index library libc.a to pick an appropriate library.

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 *TMS320C55x 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 C55X_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 will will invoke 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 C55X_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 7.4.2.2 for instructions on invoking **mklib** directly to avoid this problem.
- The index library must exist, or the linker is unable to rebuild libraries automatically.
- The index library must be in a user-writable directory, or the library is not built. If the compiler installation must be installed read-only (a good practice for shared installation), any missing libraries must be built at installation time by invoking **mklib** directly.
- The **mklib** program is specific to a certain version of a certain library; you cannot use one compiler version's run-time support's **mklib** to build a different compiler version's run-time support library.

7.4.2.2 Invoking mklib Manually

You may need to invoke **mklib** directly in special circumstances:

- The compiler installation directory is read-only or shared.
- You want to build a variant of the run-time-support library that is not pre-configured in the index library **libc.a** or known to mklib. (e.g. a variant with source-level debugging turned on.)

7.4.2.2.1 Building Standard Libraries

You can invoke mklib directly to build any or all of the libraries indexed in the index library **libc.a**. The libraries are built with the standard options for that library; the library names and the appropriate standard option sets are known to mklib.

This is most easily done by changing the working directory to be the compiler run-time-support library directory 'lib' and invoking the **mklib** executable there:

```
mklib --pattern=rts55.lib
```

7.4.2.2.2 Shared or Read-Only Library Directory

If the compiler tools are to be installed in shared or read-only directory, mklib cannot build the standard libraries at link time; the libraries must be built before the library directory is made shared or read-only.

At installation time, the installing user must build all of the libraries which will be used by any user. To build all possible libraries, change the working directory to be the compiler RTS library directory 'lib' and invoke the mklib executable there:

```
mklib --all
```

Some targets have many libraries, so this step can take a long time. To build a subset of the libraries, invoke mklib individually for each desired library.

7.4.2.2.3 Building Libraries With Custom Options

You can build a library with any extra custom options desired. This is useful for building a debugging version of the library, or 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 rts55.lib, change the working directory to the 'lib' directory and run the command:

```
mklib --pattern=rts55.lib --name=rts55_debug.lib --install_to=$Project/Debug --extra_options="-g"
```

7.4.2.2.4 The mklib Program Option Summary

Run the following command to see the full list of options. These are described in Table 7-1.

```
mklib --help
```

Table 7-1. 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 (rtssrc.zip). 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=rts55.lib --index=$C_DIR/lib
```

To build a custom library that is just like rts55.lib, but has symbolic debugging support enabled:

```
mklib --pattern=rts55.lib --extra_options="-g" --index=$C_DIR/lib --install_to=$Project/Debug
--name=rts55_debug.lib
```

7.4.3 Extending mklib

The **mklib** API is a uniform interface that allows Code Composer Studio to build libraries without needing to know exactly what underlying mechanism is used to build it. Each library vendor (e.g. the TI compiler) provides a library-specific copy of 'mklib' in the library directory that can be invoked, which understands a standardized set of options, and understands how to build the library. This allows the linker to automatically build application-compatible versions of any vendor's library without needing to register the library in advance, as long as the vendor supports mklib.

7.4.3.1 Underlying Mechanism

The underlying mechanism can be anything the vendor desires. For the compiler run-time-support libraries, mklib is just a wrapper which knows how to unpack Makefile from rtssrc.zip and invoke gmake with the appropriate options to build each library. If necessary, mklib can be bypassed and Makefile used directly, but this mode of operation is not supported by TI, and the you are responsible for any changes to 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.

7.4.3.2 Libraries From Other Vendors

Any vendor who wishes to distribute a library that can be rebuilt automatically by the linker must provide:

- An index library (like 'libc.a', but with a different name)
- A copy of mklib specific to that library
- A copy of the library source code (in whatever format is convenient)

These things must be placed together in one directory that is part of the linker's library search path (specified either in C55X_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 7-1 without error, even if they do not do anything.

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 diagnostics, it can be difficult to associate a mangled name with its corresponding name in the C++ source code. The C++ name demangler is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code.

These topics tell you how to invoke and use the C++ name demangler. The C++ name demangler reads in input, looking for mangled names. All unmangled text is copied to output unaltered. All mangled names are demangled before being copied to output.

Topic **Page**

8.1 Invoking the C++ Name Demangler

The syntax for invoking the C++ name demangler is:

dem55 [*options*] [*filenames*]

dem55 Command that invokes the C++ name demangler.

options Options affect how the name demangler behaves. Options can appear anywhere on the command line. (Options are discussed in Section 8.2.)

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, dem55 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.

8.2 C++ Name Demangler Options

The following options apply only to the C++ name demangler:

-h Prints a help screen that provides an online summary of the C++ name demangler options

-o *file* Outputs to the given file rather than to standard out

-u Specifies that external names do not have a C++ prefix

-v Enables verbose mode (outputs a banner)

8.3 Sample Usage of the C++ Name Demangler

The examples in this section illustrate the demangling process. Example 8-1 shows a sample C++ program. Example 8-2 shows the resulting assembly that is output by the compiler. In this example, the linknames of all the functions are mangled; that is, their signature information is encoded into their names.

Example 8-1. C++ Code for calories_in_a_banana

```
class banana {

public:

    int calories(void);

    banana();

    ~banana();

};

int calories_in_a_banana(void)
{
    banana x;

    return x.calories();
}
```

Example 8-2. Resulting Assembly for calories_in_a_banana

```
_calories_in_a_banana__Fv:
            AADD #-3, SP
            MOV SP, AR0
            AMAR *AR0+
            CALL #___ct__6bananaFv
            MOV SP, AR0
            AMAR *AR0+
            CALL #_calories__6bananaFv
            MOV SP, AR0
            MOV T0, *SP(#0)
            MOV #2, T0
            AMAR *AR0+
            CALL #___dt__6bananaFv
            MOV *SP(#0), T0
            AADD #3, SP
            RET
```

Executing the C++ name demangler demangles all names that it believes to be mangled. Enter:

```
dem55 calories_in_a_banana.asm
```

The result is shown in Example 8-3. The linknames in Example 8-2 ___ct__6bananaFv, _calories__6bananaFv, and ___dt__6bananaFv are demangled.

Example 8-3. Result After Running the C++ Name Demangler

```
_calories_in_a_banana():
            AADD #-3, SP
            MOV SP, AR0
            AMAR *AR0+
            CALL #banana::banana()
            MOV SP, AR0
            AMAR *AR0+
            CALL #banana::_calories()
            MOV SP, AR0
            MOV T0, *SP(#0)
            MOV #2, T0
            AMAR *AR0+
            CALL #banana::~banana()
            MOV *SP(#0), T0
            AADD #3, SP
            RET
```

Appendix A

SPRU281G–December 2011

Glossary

absolute lister— A debugging tool that allows you to create assembler listings that contain absolute 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.

alias disambiguation— A technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

aliasing— The ability for a single object to be accessed in more than one way, such as when two pointers point to a single object. It can disrupt optimization, because any indirect reference could refer to any other object.

allocation— A process in which the linker calculates the final memory addresses of output sections.

ANSI— American National Standards Institute; an organization that establishes standards voluntarily followed by industries.

archive library— A collection of individual files grouped into a single file by the archiver.

archiver— A software program that collects several individual files into a single file called an archive library. With the archiver, you can add, delete, extract, or replace members of the archive library.

assembler— A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro definitions. The assembler substitutes absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

assignment statement— A statement that initializes a variable with a value.

autoinitialization— The process of initializing global C variables (contained in the .cinit section) before program execution begins.

autoinitialization at run time— An autoinitialization method used by the linker when linking C code. The linker uses this method when you invoke it with the --rom_model link option. The linker loads the .cinit section of data tables into memory, and variables are initialized at run time.

big endian— An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

block— A set of statements that are grouped together within braces and treated as an entity.

.bss section— One of the default object file sections. You use the assembler .bss directive to reserve a specified amount of space in the memory map that you can use later for storing data. The .bss section is uninitialized.

byte— Per ANSI/ISO C, the smallest addressable unit that can hold a character.

C/C++ compiler— A software program that translates C source statements into assembly language source statements.

code generator— A compiler tool that takes the file produced by the parser or the optimizer and produces an assembly language source file.

COFF— Common object file format; a system of object files configured according to a standard developed by AT&T. These files are relocatable in memory space.

command file— A file that contains options, filenames, directives, or commands for the linker or hex conversion utility.

comment— A source statement (or portion of a source statement) that documents or improves readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

compiler program— A utility that lets you compile, assemble, and optionally link in one step. The compiler runs one or more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

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 TMS320C55x operation.

entry point— A point in target memory where execution starts.

environment variable— A system symbol that you define and assign to a string. Environmental variables are often included in Windows batch files or UNIX shell scripts such as .cshrc or .profile.

epilog— The portion of code in a function that restores the stack and returns.

executable object file— A linked, executable object file that is downloaded and executed on a target system.

expression— A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.

external symbol— A symbol that is used in the current program module but defined or declared in a different program module.

file-level optimization— A level of optimization where the compiler uses the information that it has about the entire file to optimize your code (as opposed to program-level optimization, where the compiler uses information that it has about the entire program to optimize your code).

function inlining— The process of inserting code for a function at the point of call. This saves the overhead of a function call and allows the optimizer to optimize the function in the context of the surrounding code.

global symbol— A symbol that is either defined in the current module and accessed in another, or accessed in the current module but defined in another.

high-level language debugging— The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.

indirect call— A function call where one function calls another function by giving the address of the called function.

initialization at load time— An autoinitialization method used by the linker when linking C/C++ code. The linker uses this method when you invoke it with the --ram_model link option. This method initializes variables at load time instead of run time.

initialized section— A section from an object file that will be linked into an executable object file.

input section— A section from an object file that will be linked into an executable object file.

integrated preprocessor— A C/C++ preprocessor that is merged with the parser, allowing for faster compilation. Stand-alone preprocessing or preprocessed listing is also available.

interlist feature— A feature that inserts as comments your original C/C++ source statements into the assembly language output from the assembler. The C/C++ statements are inserted next to the equivalent assembly instructions.

intrinsics— Operators that are used like functions and produce assembly language code that would otherwise be inexpressible in C, or would take greater time and effort to code.

ISO— International Organization for Standardization; a worldwide federation of national standards bodies, which establishes international standards voluntarily followed by industries.

kernel— The body of a software-pipelined loop between the pipelined-loop prolog and the pipelined-loop epilog.

K&R C— Kernighan and Ritchie C, the de facto standard as defined in the first edition of *The C Programming Language* (K&R). Most K&R C programs written for earlier, non-ISO C compilers should correctly compile and run without modification.

label— A symbol that begins in column 1 of an assembler source statement and corresponds to the address of that statement. A label is the only assembler statement that can begin in column 1.

linker— A software program that combines object files to form an executable object file that can be allocated into system memory and executed by the device.

listing file— An output file, created by the assembler, that lists source statements, their line numbers, and their effects on the section program counter (SPC).

little endian— An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

loader— A device that places an executable object file into system memory.

loop unrolling— An optimization that expands small loops so that each iteration of the loop appears in your code. Although loop unrolling increases code size, it can improve the performance of your code.

macro— A user-defined routine that can be used as an instruction.

macro call— The process of invoking a macro.

macro definition— A block of source statements that define the name and the code that make up a macro.

macro expansion— The process of inserting source statements into your code in place of a macro call.

map file— An output file, created by the linker, that shows the memory configuration, section composition, section allocation, symbol definitions and the addresses at which the symbols were defined for your program.

memory map— A map of target system memory space that is partitioned into functional blocks.

name mangling— A compiler-specific feature that encodes a function name with information regarding the function's arguments return types.

object file— An assembled or linked file that contains machine-language object code.

object library— An archive library made up of individual object files.

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.

pipelining— A technique where a second instruction begins executing before the first instruction has been completed. You can have several instructions in the pipeline, each at a different processing stage.

pop— An operation that retrieves a data object from a stack.

pragma— A preprocessor directive that provides directions to the compiler about how to treat a particular statement.

preprocessor— A software tool that interprets macro definitions, expands macros, interprets header files, interprets conditional compilation, and acts upon preprocessor directives.

program-level optimization— An aggressive level of optimization where all of the source files are compiled into one intermediate file. Because the compiler can see the entire program, several optimizations are performed with program-level optimization that are rarely applied during file-level optimization.

prolog— The portion of code in a function that sets up the stack.

push— An operation that places a data object on a stack for temporary storage.

quiet run— An option that suppresses the normal banner and the progress information.

raw data— Executable code or initialized data in an output section.

relocation— A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

run-time environment— The run time parameters in which your program must function. These parameters are defined by the memory and register conventions, stack organization, function call conventions, and system initialization.

run-time-support functions— Standard ISO functions that perform tasks that are not part of the C language (such as memory allocation, string conversion, and string searches).

run-time-support library— A library file, rts.src, that contains the source for the run time-support functions.

section— A relocatable block of code or data that ultimately will be contiguous with other sections in the memory map.

sign extend— A process that fills the unused MSBs of a value with the value's sign bit.

simulator— A software development system that simulates TMS320C55x operation.

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 a simulator or an emulator.

target system— The system on which the object code you have developed is executed.

.text section— One of the default object file sections. The .text section is initialized and contains executable code. You can use the .text directive to assemble code into the .text section.

trigraph sequence— A 3-character sequence that has a meaning (as defined by the ISO 646-1983 Invariant Code Set). These characters cannot be represented in the C character set and are expanded to one character. For example, the trigraph ??' is expanded to ^.

trip count— The number of times that a loop executes before it terminates.

unconfigured memory— Memory that is not defined as part of the memory map and cannot be loaded with code or data.

uninitialized section— A object file section that reserves space in the memory map but that has no actual contents. These sections are built with the .bss and .usect directives.

unsigned value— A value that is treated as a nonnegative number, regardless of its actual sign.

variable— A symbol representing a quantity that can assume any of a set of values.

word— A 16-bit addressable location in target memory

IMPORTANT NOTICE

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

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

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

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

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

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

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

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

Products		**Applications**	
Audio	www.ti.com/audio	Communications and Telecom	www.ti.com/communications
Amplifiers	amplifier.ti.com	Computers and Peripherals	www.ti.com/computers
Data Converters	dataconverter.ti.com	Consumer Electronics	www.ti.com/consumer-apps
DLP® Products	www.dlp.com	Energy and Lighting	www.ti.com/energy
DSP	dsp.ti.com	Industrial	www.ti.com/industrial
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Security	www.ti.com/security
Logic	logic.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Power Mgmt	power.ti.com	Transportation and Automotive	www.ti.com/automotive
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
		TI E2E Community Home Page	e2e.ti.com

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