ARM Optimizing C/C++ Compiler v20.2.0.LTS

User's Guide

Literature Number: SPNU151W
JANUARY 1998 – REVISED MARCH 2023

Table of Contents

Read This First 9
About This Manual 9
Notational Conventions 9
Related Documentation 10
Related Documentation From Texas Instruments 10
Trademarks 11
1 Introduction to the Software Development Tools 13
1.1 Software Development Tools Overview 14
1.2 Compiler Interface 15
1.3 ANSI/ISO Standard 15
1.4 Output Files 16
1.5 Utilities 16
2 Using the C/C++ Compiler 17
2.1 About the Compiler 18
2.2 Invoking the C/C++ Compiler 18
2.3 Changing the Compiler's Behavior with Options 19
2.3.1 Linker Options 25
2.3.2 Frequently Used Options 27
2.3.3 Miscellaneous Useful Options 28
2.3.4 Run-Time Model Options 29
2.3.5 Symbolic Debugging and Profiling Options 31
2.3.6 Specifying Filenames 31
2.3.7 Changing How the Compiler Interprets Filenames 32
2.3.8 Changing How the Compiler Processes C Files 32
2.3.9 Changing How the Compiler Interprets and Names Extensions 32
2.3.10 Specifying Directories 33
2.3.11 Assembler Options 33
2.3.12 Deprecated Options 34
2.4 Controlling the Compiler Through Environment Variables 34
2.4.1 Setting Default Compiler Options (TI_ARM_C_OPTION) 34
2.4.2 Naming One or More Alternate Directories (TI_ARM_C_DIR) 35
2.5 Controlling the Preprocessor 35
2.5.1 Predefined Macro Names 35
2.5.2 The Search Path for #include Files 39
2.5.3 Support for the #warning and #warn Directives 40
2.5.4 Generating a Preprocessed Listing File (--preproc_only Option) 40
2.5.5 Continuing Compilation After Preprocessing (--preproc_with_compile Option) 41
2.5.6 Generating a Preprocessed Listing File with Comments (--preproc_with_comment Option) 41
2.5.7 Generating Preprocessed Listing with Line-Control Details (--preproc_with_line Option) 41
2.5.8 Generating Preprocessed Output for a Make Utility (--preproc_dependency Option) 41
2.5.9 Generating a List of Files Included with #include (--preproc_includes Option) 41
2.5.10 Generating a List of Macros in a File (--preproc_macros Option) 41
2.6 Passing Arguments to main() 41
2.7 Understanding Diagnostic Messages 42
2.7.1 Controlling Diagnostic Messages 43
2.7.2 How You Can Use Diagnostic Suppression Options 44
2.8 Other Messages 45
2.9 Generating Cross-Reference Listing Information (--gen_cross_reference_listing Option) 45
2.10 Generating a Raw Listing File (--gen_preprocessor_listing Option) 46
2.11 Using Inline Function Expansion 47

2.11.1 Inlining Intrinsic Operators ... 48
2.11.2 Inlining Restrictions ... 48
2.12 Using Interlist ... 49
2.13 Controlling Application Binary Interface ... 49
2.14 VFP Support ... 50
2.15 Enabling Entry Hook and Exit Hook Functions ... 51
3 Optimizing Your Code ... 53
3.1 Invoking Optimization ... 54
3.2 Controlling Code Size Versus Speed ... 55
3.3 Performing File-Level Optimization (--opt_level=3 option) ... 55
3.3.1 Creating an Optimization Information File (--gen_opt_info Option) ... 55
3.4 Program-Level Optimization (--program_level_compile and --opt_level=3 options) ... 56
3.4.1 Controlling Program-Level Optimization (--call_assumptions Option) ... 56
3.4.2 Optimization Considerations When Mixing C/C++ and Assembly ... 57
3.5 Automatic Inline Expansion (--auto_inline Option) ... 58
3.6 Link-Time Optimization (--opt_level=4 Option) ... 59
3.6.1 Option Handling ... 59
3.6.2 Incompatible Types ... 59
3.7 Using Feedback Directed Optimization ... 60
3.7.1 Feedback Directed Optimization ... 60
3.7.2 Profile Data Decoder ... 62
3.7.3 Feedback Directed Optimization API ... 62
3.7.4 Feedback Directed Optimization Summary ... 62
3.8 Using Profile Information to Analyze Code Coverage ... 63
3.8.1 Code Coverage ... 63
3.8.2 Related Features and Capabilities ... 64
3.9 Accessing Aliased Variables in Optimized Code ... 65
3.10 Use Caution With asm Statements in Optimized Code ... 65
3.11 Using the Interlist Feature With Optimization ... 65
3.12 Debugging and Profiling Optimized Code ... 66
3.12.1 Profiling Optimized Code ... 66
3.13 What Kind of Optimization Is Being Performed? ... 67
3.13.1 Cost-Based Register Allocation ... 67
3.13.2 Alias Disambiguation ... 67
3.13.3 Branch Optimizations and Control-Flow Simplification ... 67
3.13.4 Data Flow Optimizations ... 68
3.13.5 Expression Simplification ... 68
3.13.6 Inline Expansion of Functions ... 68
3.13.7 Function Symbol Aliasing ... 68
3.13.8 Induction Variables and Strength Reduction ... 69
3.13.9 Loop-Invariant Code Motion ... 69
3.13.10 Loop Rotation ... 69
3.13.11 Instruction Scheduling ... 69
3.13.12 Tail Merging ... 69
3.13.13 Autoincrement Addressing ... 69
3.13.14 Block Conditionalizing ... 70
3.13.15 Epilog Inlining ... 70
3.13.16 Removing Comparisons to Zero ... 70
3.13.17 Integer Division With Constant Divisor ... 70
3.13.18 Branch Chaining ... 71
4 Linking C/C++ Code ... 73
4.1 Invoking the Linker Through the Compiler (-z Option) ... 74
4.1.1 Invoking the Linker Separately ... 74
4.1.2 Invoking the Linker as Part of the Compile Step ... 75
4.1.3 Disabling the Linker (--compile_only Compiler Option) ... 75
4.2 Linker Code Optimizations ... 76
4.2.1 Generate List of Dead Functions (--generate_dead_funcs_list Option) ... 76
4.2.2 Generating Aggregate Data Subsections (--gen_data_subsections Compiler Option) ... 76
4.3 Controlling the Linking Process ... 77
4.3.1 Including the Run-Time-Support Library ... 77
4.3.2 Run-Time Initialization ... 78

4.3.3 Initialization of Cinit and Watchdog Timer Hold ... 78
4.3.4 Global Object Constructors ... 78
4.3.5 Specifying the Type of Global Variable Initialization ... 78
4.3.6 Specifying Where to Allocate Sections in Memory ... 79
4.3.7 A Sample Linker Command File ... 80
5 C/C++ Language Implementation ... 81
5.1 Characteristics of ARM C ... 82
5.1.1 Implementation-Defined Behavior ... 83
5.2 Characteristics of ARM C++ ... 87
5.3 Using MISRA C 2004 ... 88
5.4 Using the ULP Advisor ... 89
5.5 Data Types ... 90
5.5.1 Size of Enum Types ... 91
5.6 File Encodings and Character Sets ... 92
5.7 Keywords ... 92
5.7.1 The const Keyword ... 92
5.7.2 The __interrupt Keyword ... 93
5.7.3 The volatile Keyword ... 94
5.8 C++ Exception Handling ... 95
5.9 Register Variables and Parameters ... 95
5.9.1 Local Register Variables and Parameters ... 95
5.9.2 Global Register Variables ... 96
5.10 The __asm Statement ... 97
5.11 Pragma Directives ... 98
5.11.1 The CALLS Pragma ... 99
5.11.2 The CHECK_MISRA Pragma ... 99
5.11.3 The CHECK_ULP Pragma ... 99
5.11.4 The CODE_SECTION Pragma ... 100
5.11.5 The CODE_STATE Pragma ... 100
5.11.6 The DATA_ALIGN Pragma ... 101
5.11.7 The DATA_SECTION Pragma ... 101
5.11.8 The Diagnostic Message Pragmas ... 102
5.11.9 The DUAL_STATE Pragma ... 102
5.11.10 The FORCEINLINE Pragma ... 103
5.11.11 The FORCEINLINE_RECURSIVE Pragma ... 103
5.11.12 The FUNC_ALWAYS_INLINE Pragma ... 104
5.11.13 The FUNC_CANNOT_INLINE Pragma ... 104
5.11.14 The FUNC_EXT_CALLED Pragma ... 105
5.11.15 The FUNCTION_OPTIONS Pragma ... 105
5.11.16 The INTERRUPT Pragma ... 106
5.11.17 The LOCATION Pragma ... 107
5.11.18 The MUST_ITERATE Pragma ... 107
5.11.19 The NOINIT and PERSISTENT Pragmas ... 109
5.11.20 The NOINLINE Pragma ... 110
5.11.21 The NO_HOOKS Pragma ... 110
5.11.22 The once Pragma ... 111
5.11.23 The pack Pragma ... 111
5.11.24 The PROB_ITERATE Pragma ... 112
5.11.25 The RESET_MISRA Pragma ... 112
5.11.26 The RESET_ULP Pragma ... 113
5.11.27 The RETAIN Pragma ... 113
5.11.28 The SET_CODE_SECTION and SET_DATA_SECTION Pragmas ... 114
5.11.29 The SWI_ALIAS Pragma ... 115
5.11.30 The TASK Pragma ... 116
5.11.31 The UNROLL Pragma ... 116
5.11.32 The WEAK Pragma ... 117
5.12 The _Pragma Operator ... 117
5.13 Application Binary Interface ... 118
5.14 ARM Instruction Intrinsics ... 118
5.15 Object File Symbol Naming Conventions (Linknames) ... 127
5.16 Changing the ANSI/ISO C/C++ Language Mode ... 128

5.16.1 C99 Support (--c99)....128
5.16.2 C11 Support (--c11)....129
5.16.3 Strict ANSI Mode and Relaxed ANSI Mode (--strict_ansi and --relaxed_ansi)....129
5.17 GNU , Clang, and ACLE Language Extensions....130
5.17.1 Extensions....130
5.17.2 Function Attributes....132
5.17.3 For Loop Attributes....133
5.17.4 Variable Attributes....133
5.17.5 Type Attributes....134
5.17.6 Built-In Functions....135
5.18 AUTOSAR....136
5.19 Compiler Limits....136
6 Run-Time Environment....137
6.1 Memory Model....138
6.1.1 Sections....138
6.1.2 C/C++ System Stack....139
6.1.3 Dynamic Memory Allocation....139
6.2 Object Representation....140
6.2.1 Data Type Storage....140
6.2.2 Bit Fields....144
6.2.3 Character String Constants....146
6.3 Register Conventions....147
6.4 Function Structure and Calling Conventions....149
6.4.1 How a Function Makes a Call....150
6.4.2 How a Called Function Responds....151
6.4.3 C Exception Handler Calling Convention....151
6.4.4 Accessing Arguments and Local Variables....152
6.5 Accessing Linker Symbols in C and C++....152
6.6 Interfacing C and C++ With Assembly Language....152
6.6.1 Using Assembly Language Modules With C/C++ Code....152
6.6.2 Accessing Assembly Language Functions From C/C++....153
6.6.3 Accessing Assembly Language Variables From C/C++....153
6.6.4 Sharing C/C++ Header Files With Assembly Source....155
6.6.5 Using Inline Assembly Language....155
6.6.6 Modifying Compiler Output....155
6.7 Interrupt Handling....155
6.7.1 Saving Registers During Interrupts....155
6.7.2 Using C/C++ Interrupt Routines....156
6.7.3 Using Assembly Language Interrupt Routines....156
6.7.4 How to Map Interrupt Routines to Interrupt Vectors....157
6.7.5 Using Software Interrupts....158
6.7.6 Other Interrupt Information....158
6.8 Intrinsic Run-Time-Support Arithmetic and Conversion Routines....159
6.8.1 CPSR Register and Interrupt Intrinsics....159
6.9 Built-In Functions....160
6.10 System Initialization....160
6.10.1 Boot Hook Functions for System Pre-Initialization....160
6.10.2 Run-Time Stack....161
6.10.3 Automatic Initialization of Variables....161
6.10.4 Initialization Tables....167
6.11 Dual-State Interworking Under TIABI (Deprecated)....169
6.11.1 Level of Dual-State Support....169
6.11.2 Implementation....170
7 Using Run-Time-Support Functions and Building Libraries....173
7.1 C and C++ Run-Time Support Libraries....174
7.1.1 Linking Code With the Object Library....174
7.1.2 Header Files....174
7.1.3 Modifying a Library Function....175
7.1.4 Support for String Handling....175
7.1.5 Minimal Support for Internationalization....175
7.1.6 Support for Time and Clock Functions....176

7.1.7 Allowable Number of Open Files ... 177
7.1.8 Nonstandard Header Files in the Source Tree ... 177
7.1.9 Library Naming Conventions ... 177
7.2 The C I/O Functions ... 178
7.2.1 High-Level I/O Functions ... 179
7.2.2 Overview of Low-Level I/O Implementation ... 180
7.2.3 Device-Driver Level I/O Functions ... 183
7.2.4 Adding a User-Defined Device Driver for C I/O ... 187
7.2.5 The device Prefix ... 188
7.3 Handling Reentrancy (_register_lock() and _register_unlock() Functions) ... 190
7.4 Library-Build Process ... 191
7.4.1 Required Non-Texas Instruments Software ... 191
7.4.2 Using the Library-Build Process ... 191
7.4.3 Extending mklib ... 194
8 C++ Name Demangler ... 195
8.1 Invoking the C++ Name Demangler ... 196
8.2 Sample Usage of the C++ Name Demangler ... 197
A Glossary ... 199
A.1 Terminology ... 199
B Revision History ... 206

List of Figures

Figure 1-1. ARM Software Development Flow ... 14
Figure 6-1. Char and Short Data Storage Format ... 141
Figure 6-2. 32-Bit Data Storage Format ... 142
Figure 6-3. Double-Precision Floating-Point Data Storage Format ... 143
Figure 6-4. Bit-Field Packing in Big-Endian and Little-Endian Formats ... 145
Figure 6-5. Use of the Stack During a Function Call ... 150
Figure 6-6. Autoinitialization at Run Time ... 162
Figure 6-7. Initialization at Load Time ... 166
Figure 6-8. Constructor Table ... 166
Figure 6-9. Format of Initialization Records in the .cinit Section ... 167
Figure 6-10. Format of Initialization Records in the .pinit Section ... 168

List of Tables

Table 2-1. Processor Options ... 19
Table 2-2. Optimization Options(1) ... 19
Table 2-3. Advanced Optimization Options(1) ... 20
Table 2-4. Debug Options ... 20
Table 2-5. Include Options ... 20
Table 2-6. ULP Advisor Options ... 20
Table 2-7. Control Options ... 20
Table 2-8. Language Options ... 21
Table 2-9. Parser Preprocessing Options ... 21
Table 2-10. Predefined Macro Options ... 22
Table 2-11. Diagnostic Message Options ... 22
Table 2-12. Supplemental Information Options ... 22
Table 2-13. Run-Time Model Options ... 22
Table 2-14. Entry/Exit Hook Options ... 23
Table 2-15. Feedback Options ... 23
Table 2-16. Assembler Options ... 23
Table 2-17. File Type Specifier Options ... 23
Table 2-18. Directory Specifier Options ... 24
Table 2-19. Default File Extensions Options ... 24
Table 2-20. Command Files Options ... 24
Table 2-21. MISRA-C 2004 Options ... 24
Table 2-22. Linker Basic Options ... 25
Table 2-23. File Search Path Options ... 25
Table 2-24. Command File Preprocessing Options ... 25
Table 2-25. Diagnostic Message Options ... 25

Table 2-26. Linker Output Options....26
Table 2-27. Symbol Management Options....26
Table 2-28. Run-Time Environment Options....26
Table 2-29. Miscellaneous Options....27
Table 2-30. Predefined ARM Macro Names....35
Table 2-31. ACLE Pre-Defined Macros....37
Table 2-32. Raw Listing File Identifiers....46
Table 2-33. Raw Listing File Diagnostic Identifiers....46
Table 3-1. Options That You Can Use With --opt_level=3....55
Table 3-2. Selecting a Level for the --gen_opt_info Option....55
Table 3-3. Selecting a Level for the --call_assumptions Option....56
Table 3-4. Special Considerations When Using the --call_assumptions Option....57
Table 4-1. Initialized Sections Created by the Compiler....79
Table 4-2. Uninitialized Sections Created by the Compiler....79
Table 5-1. ARM C/C++ Data Types....90
Table 5-2. Enumerator Types....90
Table 5-3. ARM Intrinsic Support by Target....118
Table 5-4. ARM Compiler Intrinsics....121
Table 5-5. GCC Language Extensions....130
Table 6-1. Summary of Sections and Memory Placement....139
Table 6-2. Data Representation in Registers and Memory....140
Table 6-3. How Register Types Are Affected by the Conventions....147
Table 6-4. Register Usage....147
Table 6-5. VFP Register Usage....148
Table 6-6. Neon Register Usage....149
Table 6-7. CPSR and Interrupt C/C++ Compiler Intrinsics....159
Table 6-8. Selecting a Level of Dual-State Support....169
Table 7-1. Differences between __time32_t and __time64_t....176
Table 7-2. The mklib Program Options....193

Read This First

About This Manual

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

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

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

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

Notational Conventions

This document uses the following conventions:

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

```
#include <stdio.h>
main()
{    printf("Hello World\n");
}
```

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

armcl [*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:

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

- In assembler syntax statements, the leftmost column is reserved for the first character of a label or symbol. If the label or symbol is optional, it is usually not shown. If a label or symbol is a required parameter, it is shown

starting against the left margin of the box, as in the example below. No instruction, command, directive, or parameter, other than a symbol or label, can begin in the leftmost column.

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

- Some directives can have a varying number of parameters. For example, the .byte directive. This syntax is shown as [, ..., *parameter*].
- The ARM® 16-bit instruction set is referred to as 16-BIS.
- The ARM 32-bit instruction set is referred to as 32-BIS.

Related Documentation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ARM C Language Extensions (ACLE) specification (ACLE Version ACLE Q2 2017)

Related Documentation From Texas Instruments

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

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

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

SPNU118 ***ARM Assembly Language Tools User's Guide.*** Describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the ARM devices.

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

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

Trademarks

This page intentionally left blank.

Chapter 1

Introduction to the Software Development Tools

The ARM® 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 *ARM Assembly Language Tools User's Guide*.

1.1 Software Development Tools Overview 14
1.2 Compiler Interface 15
1.3 ANSI/ISO Standard 15
1.4 Output Files 16
1.5 Utilities 16

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.

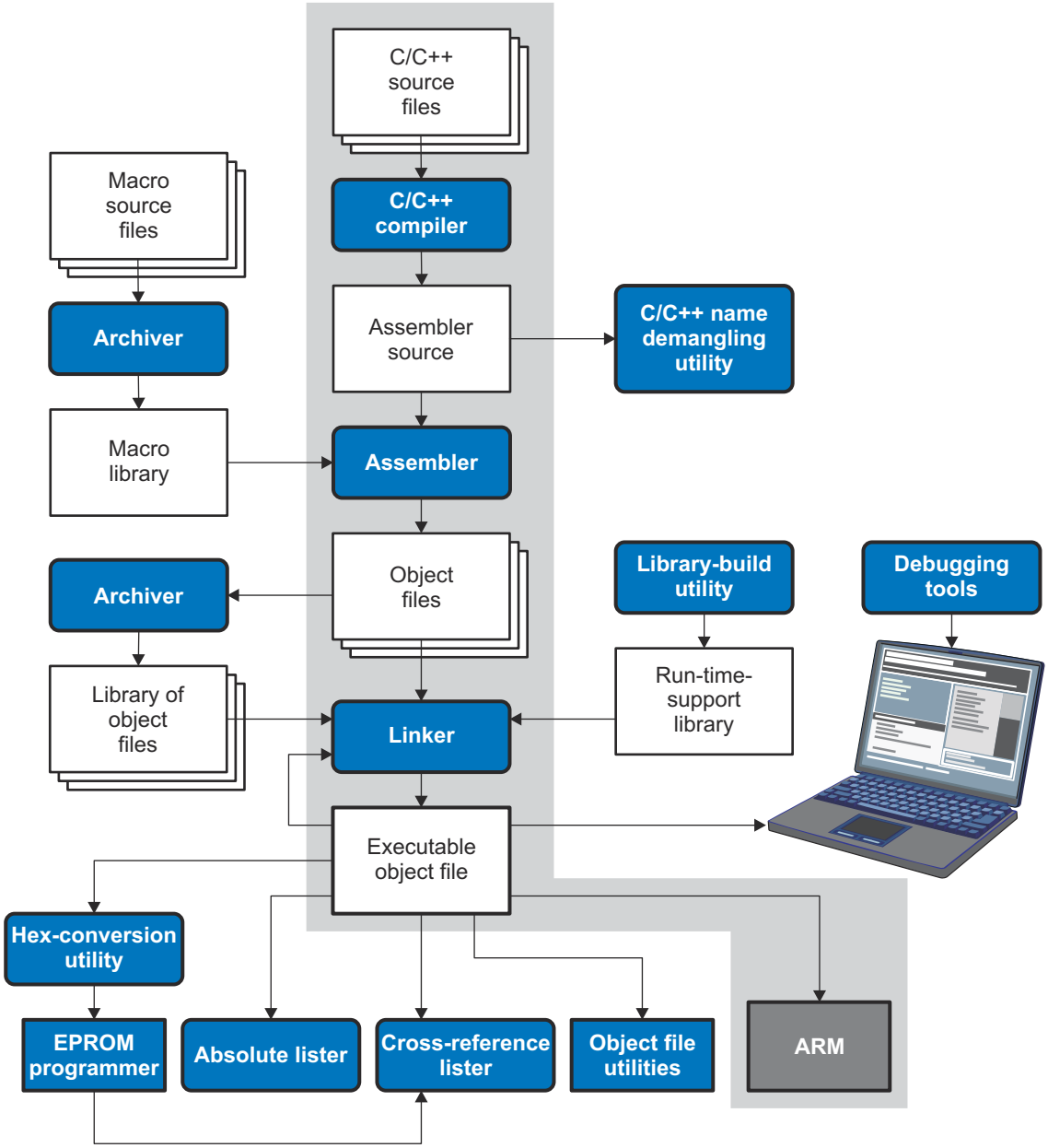


Figure 1-1. ARM Software Development Flow

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

- The **compiler** accepts C/C++ source code and produces ARM assembly language source code. See Chapter 2.
- The **assembler** translates assembly language source files into machine language relocatable object files. See the *ARM Assembly Language Tools User's Guide*.
- The **linker** combines relocatable object files into a single absolute executable object file. As it creates the executable file, it performs relocation and resolves external references. The linker accepts relocatable object

files and object libraries as input. See Chapter 4 for an overview of the linker. See the *ARM Assembly Language Tools User's Guide* for details.

- The **archiver** allows you to collect a group of files into a single archive file, called a *library*. The archiver allows you to modify such libraries by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object files. See the *ARM Assembly Language Tools User's Guide*.
- The **run-time-support libraries** contain the standard ISO C and C++ library functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the compiler. See Chapter 7.

  The **library-build utility** automatically builds the run-time-support library if compiler and linker options require a custom version of the library. See Section 7.4. Source code for the standard run-time-support library functions for C and C++ is provided in the lib\src subdirectory of the directory where the compiler is installed.
- The **hex conversion utility** converts an object file into other object formats. You can download the converted file to an EPROM programmer. See the *ARM Assembly Language Tools User's Guide*.
- The **absolute lister** accepts linked object files as input and creates .abs files as output. You can assemble these .abs files to produce a listing that contains absolute, rather than relative, addresses. Without the absolute lister, producing such a listing would be tedious and would require many manual operations. See the *ARM Assembly Language Tools User's Guide*.
- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files. See the *ARM Assembly Language Tools User's Guide*.
- The **C++ name demangler** is a debugging aid that converts names mangled by the compiler back to their original names as declared in the C++ source code. As shown in Figure 1-1, you can use the C++ name demangler on the assembly file that is output by the compiler; you can also use this utility on the assembler listing file and the linker map file. See Chapter 8.
- The **disassembler** decodes object files to show the assembly instructions that they represent. See the *ARM Assembly Language Tools User's Guide*.
- The main product of this development process is an executable object file that can be executed on a **ARM** device.

1.2 Compiler Interface

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

For more information about compiling a program, see Section 2.1.

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

1.3 ANSI/ISO Standard

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

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

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

- **ISO-standard C++:** The compiler uses the C++14 version of the C++ standard. Previously, C++03 was used. See the C++ Standard ISO/IEC 14882:2014. For a description of *unsupported* C++ features, see Section 5.2.
- **ISO-standard run-time support:** The compiler tools come with an extensive run-time library. Library functions conform to the ISO C/C++ library standard unless otherwise stated. The library includes functions for standard input and output, string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, and exponential and hyperbolic functions. Functions for signal handling are not included, because these are target-system specific. For more information, see Chapter 7.

See Section 5.16 for command line options to select the C or C++ standard your code uses.

1.4 Output Files

The following types of output files are created by the compiler:

- **ELF object files.** Executable and Linking Format (ELF) enables supporting modern language features like early template instantiation and exporting inline functions. The ELF format for ARM is part of the Application Binary Interface (ABI) specification, which is documented in the ARM Infocenter.

COFF object files and the legacy TIABI and TI ARM9 ABI modes are not supported in v15.6.0.STS and later versions of the TI Code Generation Tools. If you would like to produce COFF output files, please use v5.2 of the ARM Code Generation Tools and refer to SPNU151J for documentation.

1.5 Utilities

These features are compiler utilities:

- **Library-build utility**

  The library-build utility lets you custom-build object libraries from source for any combination of run-time models. For more information, see Section 7.4.
- **C++ name demangler**

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

Chapter 2

Using the C/C++ Compiler

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

2.1 About the Compiler 18
2.2 Invoking the C/C++ Compiler 18
2.3 Changing the Compiler's Behavior with Options 19
2.4 Controlling the Compiler Through Environment Variables 34
2.5 Controlling the Preprocessor 35
2.6 Passing Arguments to main() 41
2.7 Understanding Diagnostic Messages 42
2.8 Other Messages 45
2.9 Generating Cross-Reference Listing Information (--gen_cross_reference_listing Option) 45
2.10 Generating a Raw Listing File (--gen_preprocessor_listing Option) 46
2.11 Using Inline Function Expansion 47
2.12 Using Interlist 49
2.13 Controlling Application Binary Interface 49
2.14 VFP Support 50
2.15 Enabling Entry Hook and Exit Hook Functions 51

2.1 About the Compiler

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

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

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

Note

Invoking the Linker

By default, the compiler does not invoke the linker. You can invoke the linker by using the --run_linker (-z) compiler option. See Section 4.1.1 for details.

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

2.2 Invoking the C/C++ Compiler

To invoke the compiler, enter:

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

armcl	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 2-7 through Table 2-29.
filenames	One or more C/C++ source filesand assembly language source files.
--run_linker (**-z**)	Option that invokes the linker. The --run_linker option's short form is -z. See Chapter 4 for more information.
link_options	Options that control the linking process.
object files	Names of the object files for the linking process.

The arguments to the compiler are of three types:

- Compiler options
- Link options
- Filenames

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

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

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

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

2.3 Changing the Compiler's Behavior with Options

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

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

The following apply to the compiler options:

- There are typically two ways of specifying a given option. The "long form" uses a two hyphen prefix and is usually a more descriptive name. The "short form" uses a single hyphen prefix and a combination of letters and numbers that are not always intuitive.
- Options are usually case sensitive.
- Individual options cannot be combined.
- An option with a parameter should be specified with an equal sign before the parameter to clearly associate the parameter with the option. For example, the option to undefine a constant can be expressed as --undefine=*name*. Likewise, the option to specify the maximum amount of optimization can be expressed as -O=3. You can also specify a parameter directly after certain options, for example -O3 is the same as -O=3. No space is allowed between the option and the optional parameter, so -O 3 is not accepted.
- Files and options except the --run_linker option can occur in any order. The --run_linker option must follow all compiler options and precede any linker options.

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

Table 2-1 through Table 2-29 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={ 4 | 5e | 6 | 6M0 | 7A8 | 7M3 | 7M4 | 7R4 | 7R5 }	-mv	Selects processor version: ARM V4 (ARM7), ARM V5e (ARM9E), ARM V6 (ARM11), ARM V6M0 (Cortex-M0), ARM V7A8 (Cortex-A8), ARM V7M3 (Cortex-M3), ARM V7M4 (Cortex-M4), ARM V7R4 (Cortex-R4), or ARM V7R5 (Cortex-R5). The default is ARM V4.	Section 2.3.4
--code_state={ 16 | 32 }		Designates the ARM compilation mode.	Section 2.3.4
--float_support={ vfpv2 | vfpv3 | vfpv3d16 | fpv4spd16 | none)		Generates vector floating-point (VFP) coprocessor instructions. Use this option only if the target hardware provides this functionality.	Section 2.14
--little_endian or --endian={ big | little }	-me	Designates little-endian code. The default is big-endian.	Section 2.3.4

Table 2-2. Optimization Options(1)

Option	Alias	Effect	Section
--opt_level=off		Disables all optimization.	Section 3.1
--opt_level=*n*	-O*n*	Level 0 (-O0) optimizes register usage only. Level 1 (-O1) uses Level 0 optimizations and optimizes locally. Level 2 (-O2) uses Level 1 optimizations and optimizes globally. Level 3 (-O3) uses Level 2 optimizations and optimizes the file (default if option not used). Level 4 (-O4) uses Level 3 optimizations and performs link-time optimization.	Section 3.1, Section 3.3 , Section 3.6
--opt_for_speed[=*n*]	-mf	Controls the tradeoff between size and speed (0-5 range). If this option is specified without *n*, the default value is 4. If this option is not specified, the default setting is 1.	Section 3.2

(1) **Note:** Machine-specific options (see Table 2-13) can also affect optimization.

Table 2-3. Advanced Optimization Options(1)

Option	Alias	Effect	Section
--auto_inline=[*size*]	-oi	Sets automatic inlining size (--opt_level=3 only). If *size* is not specified, the default is 1.	Section 3.5
--call_assumptions=*n*	-op*n*	Level 0 (-op0) specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler. Level 1 (-op1) specifies that the module contains variables modified from outside the source code provided to the compiler but does not use functions called from outside the source code. Level 2 (-op2) specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler (default). Level 3 (-op3) specifies that the module contains functions that are called from outside the source code provided to the compiler but does not use variables modified from outside the source code.	Section 3.4.1
--disable_inlining		Prevents any inlining from occurring.	Section 2.11
--fp_mode={relaxed|strict}		Enables or disables relaxed floating-point mode.	Section 2.3.3
--fp_reassoc={on|off}		Enables or disables the reassociation of floating-point arithmetic.	Section 2.3.3
--gen_opt_info=*n*	-on*n*	Level 0 (-on0) disables the optimization information file. Level 1 (-on2) produces an optimization information file. Level 2 (-on2) produces a verbose optimization information file.	Section 3.3.1
--optimizer_interlist	-os	Interlists optimizer comments with assembly statements.	Section 3.11
--program_level_compile	-pm	Combines source files to perform program-level optimization.	Section 3.4
--sat_reassoc={on|off}		Enables or disables the reassociation of saturating arithmetic. Default is --sat_reassoc=off.	Section 2.3.3
--aliased_variables	-ma	Indicates that a specific aliasing technique is used.	Section 3.9

(1) **Note:** Machine-specific options (see Table 2-13) can also affect optimization.

Table 2-4. Debug Options

Option	Alias	Effect	Section
--symdebug:dwarf	-g	Default behavior. Enables symbolic debugging. The generation of debug information does not impact optimization. Therefore, generating debug information is enabled by default.	Section 2.3.5 Section 3.12
--symdebug:dwarf_version=2|3|4		Specifies the DWARF format version.	Section 2.3.5
--symdebug:none		Disables all symbolic debugging.	Section 2.3.5 Section 3.12
--symdebug:skeletal		(Deprecated; has no effect.)	

Table 2-5. Include Options

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

Table 2-6. ULP Advisor Options

Option	Alias	Effect	Section
--advice:power[={all|none|*rulespec*}]		Enables checking the specified ULP Advisor rules. (Default is all.)	Section 2.3.3
--advice:power_severity={error|warning|remark|suppress}		Sets the diagnostic severity for ULP Advisor rules.	Section 2.3.3

Table 2-7. Control Options

Option	Alias	Effect	Section
--compile_only	-c	Disables linking (negates --run_linker).	Section 4.1.3
--help	-h	Prints (on the standard output device) a description of the options understood by the compiler.	Section 2.3.2

Table 2-7. Control Options (continued)

Option	Alias	Effect	Section
--run_linker	-z	Causes the linker to be invoked from the compiler command line.	Section 2.3.2
--skip_assembler	-n	Compiles C/C++ source file , producing an assembly language output file. The assembler is not run and no object file is produced.	Section 2.3.2

Table 2-8. Language Options

Option	Alias	Effect	Section
--c89		Processes C files according to the ISO C89 standard.	Section 5.16
--c99		Processes C files according to the ISO C99 standard.	Section 5.16
--c11		Processes C files according to the ISO C11 standard.	Section 5.16
--c++14		Processes C++ files according to the ISO C++14 standard. The --c++03 option has been deprecated.	Section 5.16
--cpp_default	-fg	Processes all source files with a C extension as C++ source files.	Section 2.3.7
--enum_type={int|packed}		Choose whether to use compact integer types to store small enumerated types.	Section 2.3.4
--exceptions		Enables C++ exception handling.	Section 5.8
--extern_c_can_throw		Allow extern C functions to propagate exceptions.	--
--float_operations_allowed ={none|all|32|64}		Restricts the types of floating point operations allowed.	Section 2.3.3
--gen_cross_reference_listing	-px	Generates a cross-reference listing file (.crl).	Section 2.9
--pending_instantiations=#		Specify the number of template instantiations that may be in progress at any given time. Use 0 to specify an unlimited number.	Section 2.3.4
--plain_char={signed|unsigned}	-mc	Specifies how to treat plain chars, default is unsigned.	Section 2.3.4
--printf_support={nofloat|full| minimal}		Enables support for smaller, limited versions of the printf function family (sprintf, fprintf, etc.) and the scanf function family (sscanf, fscanf, etc.) run-time-support functions.	Section 2.3.3
--relaxed_ansi	-pr	Enables relaxed mode; ignores strict ISO violations. This is on by default. To disable this mode, use the --strict_ansi option.	Section 5.16.3
--rtti	-rtti	Enables C++ run-time type information (RTTI).	–-
--strict_ansi	-ps	Enables strict ANSI/ISO mode (for C/C++, not for K&R C). In this mode, language extensions that conflict with ANSI/ISO C/C++ are disabled. In strict ANSI/ISO mode, most ANSI/ISO violations are reported as errors. Violations that are considered discretionary may be reported as warnings instead.	Section 5.16.3
--wchar_t={32|16}		Sets the size of the C/C++ type wchar_t. Default is 16 bits.	Section 2.3.4

Table 2-9. Parser Preprocessing Options

Option	Alias	Effect	Section
--preproc_dependency[=*filename*]	-ppd	Performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility.	Section 2.5.8
--preproc_includes[=*filename*]	-ppi	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive.	Section 2.5.9
--preproc_macros[=*filename*]	-ppm	Performs preprocessing only. Writes list of predefined and user-defined macros to a file with the same name as the input but with a .pp extension.	Section 2.5.10
--preproc_only	-ppo	Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension.	Section 2.5.4
--preproc_with_comment	-ppc	Performs preprocessing only. Writes preprocessed output, keeping the comments, to a file with the same name as the input but with a .pp extension.	Section 2.5.6
--preproc_with_compile	-ppa	Continues compilation after preprocessing with any of the -pp<x> options that normally disable compilation.	Section 2.5.5

Table 2-9. Parser Preprocessing Options (continued)

Option	Alias	Effect	Section
--preproc_with_line	-ppl	Performs preprocessing only. Writes preprocessed output with line-control information (#line directives) to a file with the same name as the input but with a .pp extension.	Section 2.5.7

Table 2-10. Predefined Macro Options

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

Table 2-11. Diagnostic Message Options

Option	Alias	Effect	Section
--compiler_revision		Prints out the compiler release revision and exits.	--
--diag_error=*num*	-pdse	Categorizes the diagnostic identified by *num* as an error.	Section 2.7.1
--diag_remark=*num*	-pdsr	Categorizes the diagnostic identified by *num* as a remark.	Section 2.7.1
--diag_suppress=*num*	-pds	Suppresses the diagnostic identified by *num*.	Section 2.7.1
--diag_warning=*num*	-pdsw	Categorizes the diagnostic identified by *num* as a warning.	Section 2.7.1
--diag_wrap={on|off}		Wrap diagnostic messages (default is on). Note that this command-line option cannot be used within the Code Composer Studio IDE.	
--display_error_number	-pden	Displays a diagnostic's identifiers along with its text. Note that this command-line option cannot be used within the Code Composer Studio IDE.	Section 2.7.1
--emit_warnings_as_errors	-pdew	Treat warnings as errors.	Section 2.7.1
--gen_func_info_listing		Generate user information file (.aux).	Section 2.3.2
--issue_remarks	-pdr	Issues remarks (non-serious warnings).	Section 2.7.1
--no_warnings	-pdw	Suppresses diagnostic warnings (errors are still issued).	Section 2.7.1
--quiet	-q	Suppresses progress messages (quiet).	--
--set_error_limit=*num*	-pdel	Sets the error limit to *num*. The compiler abandons compiling after this number of errors. (The default is 100.)	Section 2.7.1
--super_quiet	-qq	Super quiet mode.	--
--tool_version	-version	Displays version number for each tool.	--
--verbose		Display banner and function progress information.	--
--verbose_diagnostics	-pdv	Provides verbose diagnostic messages that display the original source with line-wrap. Note that this command-line option cannot be used within the Code Composer Studio IDE.	Section 2.7.1
--write_diagnostics_file	-pdf	Generates a diagnostic message information file. Compiler only option. Note that this command-line option cannot be used within the Code Composer Studio IDE.	Section 2.7.1

Table 2-12. Supplemental Information Options

Option	Alias	Effect	Section
--gen_preprocessor_listing	-pl	Generates a raw listing file (.rl).	Section 2.10
--section_sizes={on|off}		Generates section size information, including sizes for sections containing executable code and constants, constant or initialized data (global and static variables), and uninitialized data. (Default is off if this option is not included on the command line. Default is on if this option is used with no value specified.)	Section 2.7.1

Table 2-13. Run-Time Model Options

Option	Alias	Effect	Section
--common={on|off}		On by default. When on, uninitialized file scope variables are emitted as common symbols. When off, common symbols are not created.	Section 2.3.4
--embedded_constants={on|off}		Controls whether compiler embeds constants in functions.	Section 2.3.4

Table 2-13. Run-Time Model Options (continued)

Option	Alias	Effect	Section
--gen_data_subsections={on|off}		Place all aggregate data (arrays, structs, and unions) into subsections. This gives the linker more control over removing unused data during the final link step. See the link to the right for details about the default setting.	Section 4.2.2
--global_register={r5|r6|r9}	-rr	Disallows use of rx=[5,6,9] by the compiler.	Section 2.3.4
-neon		Enables support for the Cortex-A8 Neon SIMD instruction set.	Section 2.3.4
--ramfunc={on|off}		If set to on, specifies that all functions should be placed in the .TI.ramfunc section, which is placed in RAM.	Section 2.3.4
--unaligned_access={on|off}		Controls generation of unaligned accesses.	Section 2.3.4
--use_dead_funcs_list [=*fname*]		Places each function listed in the file in a separate section.	Section 2.3.4

Table 2-14. Entry/Exit Hook Options

Option	Alias	Effect	Section
--entry_hook[=*name*]		Enables entry hooks.	Section 2.15
--entry_parm={none|name|address}		Specifies the parameters to the function to the --entry_hook option.	Section 2.15
--exit_hook[=*name*]		Enables exit hooks.	Section 2.15
--exit_parm={none|name|address}		Specifies the parameters to the function to the --exit_hook option.	Section 2.15
--remove_hooks_when_inlining		Removes entry/exit hooks for auto-inlined functions.	Section 2.15

Table 2-15. Feedback Options

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

Table 2-16. Assembler Options

Option	Alias	Effect	Section
--keep_asm	-k	Keeps the assembly language (.asm) file.	Section 2.3.11
--asm_listing	-al	Generates an assembly listing file.	Section 2.3.11
--c_src_interlist	-ss	Interlists C source and assembly statements.	Section 2.12 Section 3.11
--src_interlist	-s	Interlists optimizer comments (if available) and assembly source statements; otherwise interlists C and assembly source statements.	Section 2.3.2
--absolute_listing	-aa	Enables absolute listing.	Section 2.3.11
--asm_cross_reference_listing	-ax	Generates the cross-reference file.	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
--code_state={16|32}		Begins assembling instructions as 16- or 32-bit instructions.	Section 2.3.11
--include_file=*filename*	-ahi	Includes the specified file for the assembly module.	Section 2.3.11

Table 2-17. 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

Table 2-17. File Type Specifier Options (continued)

Option	Alias	Effect	Section
--cpp_file=*filename*	-fp	Identifies *filename* as a C++ file, regardless of its extension. By default, the compiler treats .C, .cpp, .cc and .cxx files as a C++ files.	Section 2.3.7
--obj_file=*filename*	-fo	Identifies *filename* as an object code file regardless of its extension. By default, the compiler and linker treat .obj files as object code files, including both *.c.obj and *.cpp.obj files.	Section 2.3.7

Table 2-18. Directory Specifier Options

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

Table 2-19. 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 2-20. Command Files Options

Option	Alias	Effect	Section
--cmd_file=*filename*	-@	Interprets contents of a file as an extension to the command line. Multiple -@ instances can be used.	Section 2.3.2

Table 2-21. 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 2.3.3
--misra_advisory={error|warning|remark|suppress}		Sets the diagnostic severity for advisory MISRA-C:2004 rules.	Section 2.3.3
--misra_required={error|warning|remark|suppress}		Sets the diagnostic severity for required MISRA-C:2004 rules.	Section 2.3.3

2.3.1 Linker Options

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

Table 2-22. Linker Basic Options

Option	Alias	Description
--run_linker	-z	Enables linking.
--output_file=*file*	-o	Names the executable output file. The default filename is a .out file.
--map_file=*file*	-m	Produces a map or listing of the input and output sections, including holes, and places the listing in *file*.
--stack_size=*size*	[-]-stack	Sets C system stack size to *size* bytes and defines a global symbol that specifies the stack size. Default = 2K 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 2-23. File Search Path Options

Option	Alias	Description
--library=*file*	-l	Names an archive library or link command *file* as linker input.
--disable_auto_rts		Disables the automatic selection of a run-time-support library. See Section 4.3.1.1.
--priority	-priority	Satisfies unresolved references by the first library that contains a definition for that symbol.
--reread_libs	-x	Forces rereading of libraries, which resolves back references.
--search_path=*pathname*	`-I`	Alters library-search algorithms to look in a directory named with *pathname* before looking in the default location. This option must appear before the --library option.

Table 2-24. 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 2-25. Diagnostic Message Options

Option	Alias	Description
--diag_error=*num*		Categorizes the diagnostic identified by *num* as an error.
--diag_remark=*num*		Categorizes the diagnostic identified by *num* as a remark.
--diag_suppress=*num*		Suppresses the diagnostic identified by *num*.
--diag_warning=*num*		Categorizes the diagnostic identified by *num* as a warning.
--display_error_number		Displays a diagnostic's identifiers along with its text.
--emit_references:file[=*file*]		Emits a file containing section information. The information includes section size, symbols defined, and references to symbols.
--emit_warnings_as_errors	-pdew	Treat warnings as errors.
--issue_remarks		Issues remarks (non-serious warnings).
--no_demangle		Disables demangling of symbol names in diagnostic messages.
--no_warnings		Suppresses diagnostic warnings (errors are still issued).
--set_error_limit=*count*		Sets the error limit to *count*. The linker abandons linking after this number of errors. (The default is 100.)
--verbose_diagnostics		Provides verbose diagnostic messages that display the original source with line-wrap.
--warn_sections	-w	Displays a message when an undefined output section is created.

Table 2-26. Linker Output Options

Option	Alias	Description
--absolute_exe	-a	Produces an absolute, executable object file. This is the default; if neither --absolute_exe nor --relocatable is specified, the linker acts as if --absolute_exe were specified.
--ecc={ on | off }		Enable linker-generated Error Correcting Codes (ECC). The default is off.
--ecc:data_error		Inject specified errors into the output file for testing.
--ecc:ecc_error		Inject specified errors into the Error Correcting Code (ECC) for testing.
--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.
--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 2-27. Symbol Management Options

Option	Alias	Description
--entry_point=*symbol*	-e	Defines a global symbol that specifies the primary entry point for the executable object file.
--globalize=*pattern*		Changes the symbol linkage to global for symbols that match *pattern*.
--hide=*pattern*		Hides symbols that match the specified *pattern*.
--localize=*pattern*		Make the symbols that match the specified *pattern* local.
--make_global=*symbol*	-g	Makes *symbol* global (overrides -h).
--make_static	-h	Makes all global symbols static.
--no_symtable	-s	Strips symbol table information and line number entries from the executable object file.
--retain		Retains a list of sections that otherwise would be discarded.
--scan_libraries	-scanlibs	Scans all libraries for duplicate symbol definitions.
--symbol_map=*refname=defname*		Specifies a symbol mapping; references to the *refname* symbol are replaced with references to the *defname* symbol. The --symbol_map option is supported when used with --opt_level=4.
--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 2-28. Run-Time Environment Options

Option	Alias	Description
--arg_size=*size*	--args	Reserve *size* bytes for the argc/argv memory area.
--cinit_hold_wdt={on|off}		Link in an RTS auto-initialization routine that either holds (on) or does not hold (off) the watchdog timer during cinit auto-initialization. See Section 4.3.3.
-be32		Forces the linker to generate BE-32 object code.
-be8		Forces the linker to generate BE-8 object code.
--cinit_compression[=type]		Specifies the type of compression to apply to the C auto initialization data. The default if this option is specified with no *type* is lzss for Lempel-Ziv-Storer-Szymanski compression.
--copy_compression[=type]		Compresses data copied by linker copy tables. The default if this option is specified with no *type* is lzss for Lempel-Ziv-Storer-Szymanski compression.
--fill_value=*value*	-f	Sets default fill value for holes within output sections
--ram_model	-cr	Initializes variables at load time. See Section 4.3.5 for details.
--rom_model	-c	Autoinitializes variables at run time. See Section 4.3.5 for details.
--trampolines[=off|on]		Generates far call trampolines (argument is optional, is "on" by default).

Table 2-29. Miscellaneous Options

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

2.3.2 Frequently Used Options

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

--c_src_interlist Invokes the interlist feature, which interweaves original C/C++ source with compiler-generated assembly language. The interlisted C statements may appear to be out of sequence. You can use the interlist feature with the optimizer by combining the --optimizer_interlist and --c_src_interlist options. See Section 3.11. The --c_src_interlist option can have a negative performance and/or code size impact.

--cmd_file=*filename* Appends the contents of a file to the option set. Use this option to avoid limitations on command line length or C style comments imposed by the operating system. Use a # or ; at the beginning of a line in the command file to include comments. You can add comments by surrounded by /* and */. To specify options, surround hyphens with quotation marks. For example, "--"quiet. You can use the --cmd_file option multiple times to specify multiple files. For example, the following indicates file3 should be compiled as source and file1 and file2 are --cmd_file files:

```
armcl --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 TI_ARM_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, *name* is set to 1. This option's short form is -D.

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

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

--gen_func_info_listing Generates a user information file with a .aux file extension. The file contains linker call graph information on a per-file level.

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

--quiet Suppresses banners and progress information from all the tools. Only source filenames and error messages are output. The --quiet option's short form is -q.

--run_linker Runs the linker on the specified object files. The --run_linker option and its parameters follow all other options on the command line. All arguments that follow --run_linker are passed to the linker. The --run_linker option's short form is -z. See Section 4.1.

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

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

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

--undefine=*name* Undefines the predefined constant *name*. This option overrides any --define options for the specified constant. The --undefine option's short form is -U.

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

2.3.3 Miscellaneous Useful Options

Following are detailed descriptions of miscellaneous options:

--advice:power={all|none|*rulespec*} Enables checking code against ULP (ultra low power) Advisor rules for possible power inefficiencies. More detailed information can be found at www.ti.com/ulpadvisor. The rulespec parameter is a comma-separated list of specifiers. See Section 5.4 for details.

--advice:power_severity={error|warning|remark|suppress} Sets the diagnostic severity for ULP Advisor rules.

--check_misra={all|required|advisory|none|*rulespec*} Displays the specified amount or type of MISRA-C documentation. This option must be used if you want to enable use of the CHECK_MISRA and RESET_MISRA pragmas within the source code. The *rulespec* parameter is a comma-separated list of specifiers. See Section 5.3 for details.

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

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

- If a double-precision floating-point expression's result is assigned to a single-precision floating-point, an integer, or immediately used in a single-precision context, the expression's computations are converted to single-precision computations. Double-precision constants in the expression are converted to single-precision if they can be correctly represented as single-precision constants.
- Calls to double-precision functions in math.h are converted to their single-precision counterparts if all arguments are single-precision and the result is used in a single-precision context. The math.h header file must be included for this optimization to work.
- Division by a constant is converted to inverse multiplication.
- Calls to sqrt, sqrtf, and sqrtl are converted directly to the VSQRT instruction. In this case errno will not be set for negative inputs.
- Certain C standard float functions--such as sqrt, sin, cos, atan, atan2, and fmodf--are redirected to optimized inline functions where possible.

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

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

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

2.3.4 Run-Time Model Options

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

The ARM compiler now supports only the Embedded Application Binary Interface (EABI) ABI, which uses the ELF object format and the DWARF debug format. If you want support for the legacy COFF ABI, please use the ARM v5.2 Code Generation Tools and refer to SPNU151J and SPNU118J for documentation.

--code_state={16|32}	Generates 16-bit Thumb code. By default, 32-bit code is generated. When Cortex-R4, Cortex-M0, Cortex-M3, or Cortex-A8 architecture support is chosen, the --code_state option generates Thumb-2 code. For details on indirect calls in 16-bit versus 32-bit code, see Section 6.11.2.2.
--common={on|off}	When on (the default), uninitialized file scope variables are emitted as common symbols. When off, common symbols are not created. The benefit of allowing common symbols to be created is that generated code can remove unused variables that would otherwise increase the size of the .bss section. (Uninitialized variables of a size larger than 32 bytes are separately optimized through placement in separate subsections that can be omitted from a link.) Variables cannot be common symbols if they are assigned to a section other than .bss or are defined relative to another common symbol.
--embedded_constants={on|off}	By default the compiler embeds constants in functions. These constants can include literals, addresses, strings, etc. This is a problem if you wants to prevent reads from a memory region that contains only executable code. To enable the generation of "execute only code", the compiler provides the --embedded_constants=[on|off] option. If the option is not specified, it is assumed to be on. The option is available on the following devices: Cortex-A8, Cortex-M3, Cortex-M4, and Cortex-R4.
--endian={ big | little }	Designates big- or little-endian format for the compiled code. By default, big-endian format is used.
--enum_type={int|packed}	Designates the underlying type of an enumeration type. The default is packed, which causes the underlying enumeration type to be the smallest integer type that accommodates the enumeration constants. Using --enum_type=int causes the underlying type to always be int. An enumeration constant with a value outside the int range generates an error.
--float_support={ vfpv2 | vfpv3 | vfpv3d16 | fpv4spd16 | none }	Generates vector floating-point (VFP) coprocessor instructions for various versions and libraries. See Section 2.14.
--global_register={r5|r6|r9}	Disallows use of rx=[5|6|9] by the compiler. Only one --global_register option may be used on a command line; if multiple such options are specified, only the last option takes effect.
-md	Disables dual-state interworking support. See Section 6.11.1.
-mv={4|5e|6|6M0|7A8|7M3 |7M4|7R4|7R5}	Selects processor version: ARM V4 (ARM7), ARM V5e (ARM9E), ARM V6 (ARM11), ARM V6M0 (Cortex-M0), ARM V7A8 (Cortex-A8), ARM V7M3 (Cortex-M3), ARM V7M4 (Cortex-M4), ARM V7R4 (Cortex-R4), or ARM V7R5 (Cortex-R5). The default is ARM V4.

--neon
The compiler can generate code using the SIMD instructions available in the Neon extension to the version 7 ARM architecture. The optimizer attempts to vectorize source code in order to take advantage of these SIMD instructions. In order to generate vectorized SIMD Neon code, select the version 7 architecture with the -mv=7A8 option and enable Neon instruction support with the --neon option.
The optimizer is used to vectorize the source code. At least level 2 optimization (--opt_level=2 or O2) is required, although level 3 (--opt_level=3) is recommended along with the --opt_for_speed option.

--pending_instantiations=#
Specify the number of template instantiations that may be in progress at any given time. Use 0 to specify an unlimited number.

--plain_char={signed|unsigned}
Specifies how to treat C/C++ plain char variables. Default is unsigned.

--ramfunc={on|off}
If set to on, specifies that all functions should be placed in the .TI.ramfunc section, which is placed in RAM. If set to off, only functions with the ramfunc function attribute are treated this way. See Section 5.17.2.
Newer TI linker command files support the --ramfunc option automatically by placing functions in the .TI.ramfunc section. If you have a linker command file that does not include a section specification for the .TI.ramfunc section, you can modify the linker command file to place this section in RAM. See the *ARM Assembly Language Tools User's Guide* for details on section placement.

--silicon_version
Selects the instruction set version. The options are:

- 4 = ARM V4 (ARM7) This is the default.
- 5e = ARM V5e (ARM9E)
- 6 = ARM V6 (ARM11)
- 6M0 = ARM V6M0 (Cortex-M0)
- 7A8 = ARM V7A8 (Cortex-A8)
- 7M3 = ARM V7M3 (Cortex-M3)
- 7M4 = ARM V7M4 (Cortex-M4)
- 7R4 = ARM V7R4 (Cortex-R4),
- 7R5 = ARM V7R5 (Cortex-R5)

Using the --silicon_version=7M4 option automatically sets the --float_support=fpv4spd16 option. To disable hardware floating point support, use the --float_support=none option.

--unaligned_access={on|off}
Informs the compiler that the target device supports unaligned memory accesses. Typically data is aligned to its size boundary. For instance 32-bit data is aligned on a 32-bit boundary, 16-bit data on a 16-bit boundary, and 8-bit data on an 8-bit boundary. If this option is set to on, it tells the compiler it is legal to generate load and store instructions for data that falls on an unaligned boundary (32-bit data on a 16-bit boundary). Cases where unaligned data accesses can occur include calls to memcpy() and accessing packed structs. This option is on by default for all Cortex devices.

--use_dead_funcs_list[=*fname*]
Places each function listed in the file in a separate section. The functions are placed in the *fname* section, if specified. This option and --generate_dead_funcs_list are not recommended within the Code Composer Studio IDE. Instead, consider using --opt_level=4, --program_level_compile, and/or --gen_func_subsections.

--wchar_t={32|16}
Sets the size (in bits) of the C/C++ type wchar_t. By default the compiler generates 16-bit wchar_t. 16-bit wchar_t objects are not compatible with the 32-bit wchar_t objects; an error is generated if they are combined.

2.3.5 Symbolic Debugging and Profiling Options

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

--symdebug:dwarf	(Default) Generates directives that are used by the C/C++ source-level debugger and enables assembly source debugging in the assembler. The --symdebug:dwarf option's short form is -g. See Section 3.12. For details on the DWARF format, see *The DWARF Debugging Standard*.
--symdebug:dwarf_version={2|3|4}	Specifies the DWARF debugging format version (2, 3, or 4) to be generated when --symdebug:dwarf (the default) is specified. By default, the compiler generates DWARF version 3 debug information. DWARF versions 2, 3, and 4 may be intermixed safely. When DWARF 4 is used, type information is placed in the .debug_types section. At link time, duplicate type information is removed. This method of type merging is superior to DWARF 2 or 3 and results in a smaller executable. In addition, DWARF 4 reduces the size of intermediate object files in comparison to DWARF 3. For more about TI extensions to the DWARF language, see *The Impact of DWARF on TI Object Files* (SPRAAB5).
--symdebug:none	Disables all symbolic debugging output. This option is not recommended; it prevents debugging and most performance analysis capabilities.
--symdebug:skeletal	Deprecated. Has no effect.

2.3.6 Specifying Filenames

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

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

Note

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

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

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

```
armcl *.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:

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

You cannot use the filename options with wildcard specifications.

Note

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

2.3.8 Changing How the Compiler Processes C Files

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

2.3.9 Changing How the Compiler Interprets and Names Extensions

You can use options to change how the compiler program interprets filename extensions and names the extensions of the files that it creates. The filename extension options must precede the filenames they apply to on the command line. You can use wildcard specifications with these options. An extension can be up to nine characters in length. Select the appropriate option for the type of extension you want to specify:

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

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

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

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

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

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

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

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

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

--temp_directory=*directory*
Specifies a directory for temporary intermediate files. For example:
```
armcl --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 *ARM Assembly Language Tools User's Guide*.

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

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

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

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

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

--asm_listing
Produces an assembly listing file.

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

--code_state={16|32}
Generates 16-bit Thumb code. By default, 32-bit code is generated. When Cortex-R4, Cortex-M0, Cortex-M3, or Cortex-A8 architecture support is chosen, the --code_state option generates Thumb-2 code. For details on indirect calls in 16-bit versus 32-bit code, see Section 6.11.2.2.

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

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

2.3.12 Deprecated Options

Several compiler options have been deprecated, removed, or renamed. The compiler continues to accept some of the deprecated options, but they are not recommended for use.

2.4 Controlling the Compiler Through Environment Variables

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

Note

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

2.4.1 Setting Default Compiler Options (TI_ARM_C_OPTION)

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

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

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

```
set TI_ARM_C_OPTION=--quiet --src_interlist --run_linker
```

Note

The TI_ARM_C_OPTION environment variable takes precedence over the older TMS470_C_OPTION environment variable if both are defined. If only TMS470_C_OPTION is set, it will continue to be used.

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

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

2.4.2 Naming One or More Alternate Directories (TI_ARM_C_DIR)

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

Note

The TI_ARM_C_DIR environment variable takes precedence over the older TMS470_C_DIR environment variable if both are defined. If only TMS470_C_DIR is set, it will continue to be used.

2.5 Controlling the Preprocessor

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

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

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

2.5.1 Predefined Macro Names

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

Table 2-30. Predefined ARM Macro Names

Macro Name	Description
__16bis__	Defined if 16-BIS state is selected (the -code_state=16 option is used); otherwise, it is undefined.
__32bis__	Defined if 32-BIS state is selected (the -code_state=16 option is not used); otherwise, it is undefined.
_AEABI_PORTABILITY_LEVEL	Define to 1 to enable full object file portability when headers files are included. Define to 0 to require full C standard compatibility. See the ARM standard for details.

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

Macro Name	Description
__big_endian__	Defined if big-endian mode is selected (the --endian=big option is used or the --endian=little option is not used); otherwise, it is undefined.
__DATE__ (1)	Expands to the compilation date in the form *mmm dd yyyy*
__FILE__ (1)	Expands to the current source filename
_INLINE	Expands to 1 if optimization is used (--opt_level or -O option); undefined otherwise.
__LINE__ (1)	Expands to the current line number
__little_endian__	Defined if little-endian mode is selected (the --endian=little option is used); otherwise, it is undefined.
__PTRDIFF_T_TYPE__	Defined to the type of ptrdiff_t.
__signed_chars__	Defined if char types are signed by default
__SIZE_T_TYPE__	Defined to the type of size_t.
__STDC__ (1)	Defined to 1 to indicate that compiler conforms to ISO C Standard. See Section 5.1 for exceptions to ISO C conformance.
__STDC_VERSION__	C standard macro.
__STDC_HOSTED__	C standard macro. Always defined to 1.
__STDC_NO_THREADS__	C standard macro. Always defined to 1.
__TI_COMPILER_VERSION__	Defined to a 7-9 digit integer, depending on if X has 1, 2, or 3 digits. The number does not contain a decimal. For example, version 3.2.1 is represented as 3002001. The leading zeros are dropped to prevent the number being interpreted as an octal.
__TI_EABI_SUPPORT__	Defined to 1 if the EABI ABI is enabled (this is the default); otherwise, it is undefined.
__TI_FPALIB_SUPPORT__	Defined to 1 if the FPA endianness is used to store double-precision floating-point values; otherwise, it is undefined.
__TI_GNU_ATTRIBUTE_SUPPORT__	Defined to 1 if GCC extensions are enabled (which is the default)
__TI_NEON_SUPPORT__	Defined to 1 if NEON SIMD extension is targeted (the --neon option is used); otherwise, it is undefined.
__TI_STRICT_ANSI_MODE__	Defined to 1 if strict ANSI/ISO mode is enabled (the --strict_ansi option is used); otherwise, it is defined as 0.
__TI_STRICT_FP_MODE__	Defined to 1 if --fp_mode=strict is used (default); otherwise, it is defined as 0.
__TI_ARM__	Always defined
__TI_ARM_V4__	Defined to 1 if the v4 architecture (ARM7) is targeted (the -mv4 option is used); otherwise, it is undefined.
__TI_ARM_V5__	Defined to 1 if the v5E architecture (ARM9E) is targeted (the -mv5e option is used); otherwise, it is undefined.
__TI_ARM_V6__	Defined to 1 if the v6 architecture (ARM11) is targeted (the -mv6 option is used); otherwise, it is undefined.
__TI_ARM_V6M0__	Defined to 1 if the v6M0 architecture (Cortex-M0) is targeted (the -mv6M0 option is used); otherwise, it is undefined.
__TI_ARM_V7__	Defined to 1 if any v7 architecture (Cortex) is targeted; otherwise, it is undefined.
__TI_ARM_V7A8__	Defined to 1 if the v7A8 architecture (Cortex-A8) is targeted (the -mv7A8 option is used); otherwise, it is undefined.
__TI_ARM_V7M3__	Defined to 1 if the v7M3 architecture (Cortex-M3) is targeted (the -mv7M3 option is used); otherwise, it is undefined.
__TI_ARM_V7M4__	Defined to 1 if the v7M4 architecture (Cortex-M4) is targeted (the -mv7M4 option is used); otherwise, it is undefined.
__TI_ARM_V7R4__	Defined to 1 if the v7R4 architecture (Cortex-R4) is targeted (the -mv7R4 option is used); otherwise, it is undefined.
__TI_ARM_V7R5__	Defined to 1 if the v7R5 architecture (Cortex-R5) is targeted (the -mv7R5 option is used); otherwise, it is undefined.
__TI_FPV4SPD16_SUPPORT__	Defined to 1 if the VFP coprocessor is enabled (the --float_support=fpv4spd16 option is used); otherwise, it is undefined.
__TI_VFP_SUPPORT__	Defined to 1 if the VFP coprocessor is enabled (any --float_support option is used); otherwise, it is undefined.

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

Macro Name	Description
__TI_VFPLIB_SUPPORT__	Defined to 1 if the VFP endianness is used to store double-precision floating-point values; otherwise, it is undefined.
__TI_VFPV3_SUPPORT__	Defined to 1 if the VFP coprocessor is enabled (the --float_support=vfpv3 option is used); otherwise, it is undefined.
__TI_VFPV3D16_SUPPORT__	Defined to 1 if the VFP coprocessor is enabled (the --float_support=vfpv3d16 option is used); otherwise, it is undefined.
__TI_WCHAR_T_BITS__	Defined to the type of wchar_t.
__TIME__ (1)	Expands to the compilation time in the form "*hh:mm:ss*"
__unsigned_chars__	Defined if char types are unsigned by default (default)
__WCHAR_T_TYPE__	Defined to the type of wchar_t.

(1) Specified by the ISO standard

Note

Macros with names that contain __TI_ARM are duplicates of the older __TI_TMS470 macros. For example, __TI_ARM_V7__ is the newer name for the __TI_TMS470_V7__ macro. The old macro names still exist and can continue to be 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");
```

In addition, the ARM C Language Extensions (ACLE) v2.0 specification describes macros that identify features of the ARM architecture and how the C/C++ implementation uses the architecture. All ACLE predefined macros begin with the prefix __ARM. Table 2-31 lists the macros mentioned in the ACLE specification and the section of the specification that provides more information. Some macros are undefined because they are not applicable for any Cortex-M or Cortex-R processor variant.

Table 2-31. ACLE Pre-Defined Macros

Macro Name	Description	Section in ACLE Specification
__ARM_32BIT_STATE	Defined as 1 if the compiler is generating code for an ARM 32-bit processor variant (-mv6m0, -mv7m3, -mv7m4, -mv7a8, -mv7r4, and -mv7r5); undefined otherwise.	(Section 5.4.1)
__ARM_64BIT_STATE	Undefined	(Section 5.4.1)
__ARM_ACLE	Defined as 200 for all Cortex-M and Cortex-R processor variants (-mv6m0, -mv7m3, -mv7m4, -mv7r4, and -mv7r5).	(Sections 3.4, 5.2)
__ARM_ALIGN_MAX_PWR	Not supported	(Section 6.5.2)
__ARM_ALIGN_MAX_STACK_PWR	Not supported	(Section 6.5.3)
__ARM_ARCH	Identifies the version of ARM architecture selected on the compiler command line. • 4 indicates -mv4 • 5 indicates -mv5e • 6 indicates -mv6 or -mv6m0 • 7 indicates -mv7a8, -mv7m3, -mv7m4, -mv7r4, or -mv7r5	(Section 5.1)
__ARM_ARCH_ISA_A64	Undefined	(Section 5.4.1)
__ARM_ARCH_ISA_ARM	Defined as 1 if the compiler is generating code for a processor variant that supports the ARM instruction set (-mv7a8, -mv7r4, and -mv7r5); undefined otherwise.	(Section 5.4.1)

Table 2-31. ACLE Pre-Defined Macros (continued)

Macro Name	Description	Section in ACLE Specification
__ARM_ARCH_ISA_THUMB	Defined as 1 if the compiler is generating code for a processor variant that supports the THUMB-1 instruction set. Defined as 2 if the compiler is generating code for a processor variant that supports the THUMB-2 instruction set; undefined otherwise.	(Section 5.4.1)
__ARM_ARCH_PROFILE	Not supported	(Section 5.4.2)
__ARM_BIG_ENDIAN	Defined as 1 by default; not defined if --little-endian (-me) option is used.	(Section 5.3)
__ARM_FEATURE_CLZ	Defined as 1 if the compiler is generating code for a processor variant that supports the CLZ instruction (-mv7m3, -mv7m4, -mv7a8, -mv7r4, and -mv7r5); undefined otherwise.	(Section 5.4.5)
__ARM_FEATURE_COPROC	Not supported	(Section 5.9)
__ARM_FEATURE_CRC32	Undefined	(Section 5.5.8)
__ARM_FEATURE_CRYPTO	Undefined	(Section 5.5.7)
__ARM_FEATURE_DIRECTED_ROUNDING	Undefined	(Section 5.5.9)
__ARM_FEATURE_DSP	Defined as 1 if the compiler is generating code for a Cortex-M or Cortex-R processor that supports DSP instructions/intrinsics (-mv7m4, -mv7r4, and -mv7r5); undefined otherwise.	(Section 5.4.7)
__ARM_FEATURE_FMA	Not supported	(Section 5.5.3)
__ARM_FEATURE_FP16_SCALAR_ ARITHMETIC	Undefined	(Sections 3.4, 5.5.13)
__ARM_FEATURE_FP16_VECTOR_ ARITHMETIC	Undefined	(Section 5.5.13)
__ARM_FEATURE_IDIV	Not supported	(Section 5.4.10)
__ARM_FEATURE_JCVT	Undefined	(Section 5.5.14)
__ARM_FEATURE_LDREX	Undefined	(Section 5.4.4)
__ARM_FEATURE_NUMERIC_MAXMIN	Undefined	(Section 5.5.10)
__ARM_FEATURE_QBIT	Not supported	(Section 5.4.6)
__ARM_FEATURE_QRDMX	Undefined	(Section 5.5.12)
__ARM_FEATURE_SAT	Defined as 1 if the compiler is generating code for a processor variant that supports SSAT/USAT instructions/ intrinsics (-mv7m3, -mv7m4, -mv7a8, -mv7r4, and -mv7r5); undefined otherwise.	(Section 5.4.8)
__ARM_FEATURE_SIMD32	Defined as 1 if the compiler is generating code for a processor variant that supports all SIMD instructions/intrinsics (-mv7m4, -mv7r4, and -mv7r5); undefined otherwise.	(Section 5.4.9)
__ARM_FEATURE_UNALIGNED	Defined as 1 if the compiler is generating code for a processor variant that supports unaligned access to memory (-mv7m3, -mv7m4, -mv7a8, -mv7r4, and -mv7r5); undefined otherwise.	(Section 5.4.3)
__ARM_FP	Defined as 6 for --float_support={fpv4spd16 | fpv5spd16}. Defined as 12 for --float_support={vfpv2 | vfpv3 | vfpv3d16}; undefined otherwise.	(Section 5.5.1)
__ARM_FP16_ARGS	Defined as 1 if a 16-bit float type can be used for an argument and/or result; undefined otherwise.	(Section 5.5.11)
__ARM_FP16_FORMAT_ALTERNATIVE	Undefined	(Section 5.5.2)
__ARM_FP16_FORMAT_IEEE	Defined as 1 if the IEEE format for 16-bit floating-point (according to IEEE 754-2008 standard) is used; undefined otherwise.	(Section 5.5.2)
__ARM_FP_FAST	Not supported	(Section 5.6)
__ARM_FP_FENV_ROUNDING	Not supported	(Section 5.6)
__ARM_NEON	Undefined	(Sections 3.4, 5.5.4)
__ARM_NEON_FP	Undefined	(Section 5.5.5)

Table 2-31. ACLE Pre-Defined Macros (continued)

Macro Name	Description	Section in ACLE Specification
__ARM_PCS	Defined as 1 if the compiler can assume the default procedure calling standard for a translation unit conforms to the "base procedure call standard" as prescribed in the ARM Architecture Procedure Call Standard (AAPCS) specification (-mv7m3, -mv7m4, -mv7r4, and -mv7r5); undefined otherwise.	(Section 5.7)
__ARM_PCS_AAPCS64	Undefined	(Section 5.7)
__ARM_PCS_VFP	Defined as 1 if the default procedure calling convention is to pass floating-point arguments / return values in hardware floating-point registers; undefined otherwise.	(Section 5.7)
__ARM_ROPI	Undefined	(Section 5.8)
__ARM_RWPI	Undefined	(Section 5.8)
__ARM_SIZEOF_MINIMAL_ENUM	Defined to the smallest possible enum type size (1 byte for packed, 4 bytes for int). This mirrors the --enum_type=[packed | int] option where packed is the default.	(Section 3.1.1)
__ARM_SIZEOF_WCHAR_T	Defined as 2 if --wchar_t=16 (default). Defined as 4 if --wchar_t=32.	(Section 3.1.1)
__ARM_WMMX	Undefined	(Section 5.5.6)
__STDC_IEC_559__	Undefined	(Section 5.6)
__SUPPORT_SNAN__	Not supported	(Section 5.6)

2.5.2 The Search Path for #include Files

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

- If you enclose the filename in double quotes (" "), the compiler searches for the file in this order:
  1. The directory of the file that contains the #include directive and in the directories of any files that contain that file.
  2. Directories named with the --include_path option.
  3. Directories set with the TI_ARM_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 TI_ARM_C_DIR environment variable.

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

2.5.2.1 Adding a Directory to the #include File Search Path (--include_path Option)

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

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

There is no limit to the number of --include_path options per invocation of the compiler; each --include_path option names one *directory*. In C source, you can use the #include directive without specifying any directory information for the file; instead, you can specify the directory information with the --include_path option.

For example, assume that a file called source.c is in the current directory. The file source.c contains the following directive statement:

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

Assume that the complete pathname for alt.h is:

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

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

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

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

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

or invoke the compiler with the following command:

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

and file.c contains this line:

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

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

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

2.5.3 Support for the #warning and #warn Directives

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

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

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

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

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

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

2.5.5 Continuing Compilation After Preprocessing (--preproc_with_compile Option)

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

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

The --preproc_with_comment option performs all of the preprocessing functions except removing comments and generates a preprocessed version of your source file with a .pp extension. Use the --preproc_with_comment option instead of the --preproc_only option if you want to keep the comments.

2.5.7 Generating Preprocessed Listing with Line-Control Details (--preproc_with_line Option)

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

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

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

2.5.9 Generating a List of Files Included with #include (--preproc_includes Option)

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

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

The --preproc_macros option generates a list of all predefined and user-defined macros. If you do not supply an optional filename, the list is written to a file with the same name as the source file but with a .pp extension.

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

2.6 Passing Arguments to main()

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

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

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

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

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

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

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

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

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

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

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

2.7 Understanding Diagnostic Messages

One of the primary functions of the compiler and linker is to report diagnostic messages for the source program. A diagnostic message indicates that something may be wrong with the program. When the compiler or linker detects a suspect condition, it displays a message in the following format:

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

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

Diagnostic messages have a severity, as follows:

- A **fatal error** indicates a problem so severe that the compilation cannot continue. Examples of such problems include command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one will not be compiled.
- An **error** indicates a violation of the syntax or semantic rules of the C/C++ language. Compilation may continue, but object code is not generated.
- A **warning** indicates something that is likely to be a problem, but cannot be proven to be an error. For example, the compiler emits a warning for an unused variable. An unused variable does not affect program execution, but its existence suggests that you might have meant to use it. Compilation continues and object code is generated (if no errors are detected).
- A **remark** is less serious than a warning. It may indicate something that is a potential problem in rare cases, or the remark may be strictly informational. Compilation continues and object code is generated (if no errors are detected). By default, remarks are not issued. Use the --issue_remarks compiler option to enable remarks.
- **Advice** provides information about recommended usage. It is not provided in the same way as the other diagnostic categories described here. Instead, it is only available in Code Composer Studio in the Advice area, which is a tab that appears next to the Problems tab. This advice cannot be controlled or accessed via the command line. The advice provided includes suggested settings for the --opt_level and --opt_for_speed options. In addition, messages about suggested code changes from the ULP (Ultra-Low Power) Advisor are provided in this tab.

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

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

By default, the source code line is not printed. Use the --verbose_diagnostics compiler option to display the source line and the error position. The above example makes use of this option.

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

Long messages are wrapped to additional lines, when necessary.

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

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

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

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

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

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

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

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

2.7.1 Controlling Diagnostic Messages

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

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

--diag_remark=*num*	Categorizes the diagnostic identified by *num* as a remark. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate compile. Then use --diag_remark=*num* to recategorize the diagnostic as a remark. You can only alter the severity of discretionary diagnostic messages.
--diag_suppress=*num*	Suppresses the diagnostic identified by *num*. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate compile. Then use --diag_suppress=*num* to suppress the diagnostic. You can only suppress discretionary diagnostic messages.
--diag_warning=*num*	Categorizes the diagnostic identified by *num* as a warning. To determine the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate compile. Then use --diag_warning=*num* to recategorize the diagnostic as a warning. You can only alter the severity of discretionary diagnostic messages.
--display_error_number	Displays a diagnostic's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (--diag_suppress, --diag_error, --diag_remark, and --diag_warning). This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix -D; otherwise, no suffix is present. See Section 2.7.
--emit_warnings_as_errors	Treats all warnings as errors. This option cannot be used with the --no_warnings option. The --diag_remark option takes precedence over this option. This option takes precedence over the --diag_warning option.
--issue_remarks	Issues remarks (non-serious warnings), which are suppressed by default.
--no_warnings	Suppresses diagnostic warnings (errors are still issued).
--section_sizes={on|off}	Generates section size information, including sizes for sections containing executable code and constants, constant or initialized data (global and static variables), and uninitialized data. Section size information is output during both the assembly and linking phases. This option should be placed on the command line with the compiler options (that is, before the --run_linker or --z option).
--set_error_limit=*num*	Sets the error limit to *num*, which can be any decimal value. The compiler abandons compiling after this number of errors. (The default is 100.)
--verbose_diagnostics	Provides verbose diagnostic messages that display the original source with line-wrap and indicate the position of the error in the source line. Note that this command-line option cannot be used within the Code Composer Studio IDE.
--write_diagnostics_file	Produces a diagnostic message information file with the same source file name with an *.err* extension. (The --write_diagnostics_file option is not supported by the linker.) Note that this command-line option cannot be used within the Code Composer Studio IDE.

2.7.2 How You Can Use Diagnostic Suppression Options

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

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

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

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

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

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

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

Note

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

2.8 Other Messages

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

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

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

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

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

sym-id name X filename line number column number

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

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

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

The raw listing file contains the following information:

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

Each source line in the raw listing file begins with one of the identifiers listed in Table 2-32.

Table 2-32. Raw Listing File Identifiers

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

The --gen_preprocessor_listing option also includes diagnostic identifiers as defined in Table 2-33.

Table 2-33. 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 2-33 that indicates the severity of the diagnostic
filename	The source file
line number	The line number in the source file
column number	The column number in the source file
diagnostic	The message text for the diagnostic

Diagnostic messages after the end of file are indicated as the last line of the file with a column number of 0. When diagnostic message text requires more than one line, each subsequent line contains the same file, line, and column information but uses a lowercase version of the diagnostic identifier. For more information about diagnostic messages, see Section 2.7.

2.11 Using Inline Function Expansion

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

Note

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

Function inlining is triggered by the following situations:

- The use of built-in intrinsic operations. Intrinsic operations look like function calls, and are inlined automatically, even though no function body exists.
- Use of the `inline` keyword or the equivalent `__inline` keyword. Functions declared with the inline keyword may be inlined by the compiler if you set --opt_level=0 or greater. The inline keyword is a suggestion from the programmer to the compiler. Even if your optimization level is high, inlining is still optional for the compiler. The compiler decides whether to inline a function based on the length of the function, the number of times it is called, your --opt_for_speed setting, and any contents of the function that disqualify it from inlining (see Section 2.11.2). Functions can be inlined at --opt_level=0 or above if the function body is visible in the same module or if -pm is also used and the function is visible in one of the modules being compiled. Functions may be inlined at link time if the file containing the definition and the call site were both compiled with --opt_level=4. Functions defined as both static and inline are more likely to be inlined.
- When --opt_level=3 or greater is used, the compiler may automatically inline eligible functions even if they are not declared as inline functions. The same list of decision factors listed for functions explicitly defined with the inline keyword is used. For more about automatic function inlining, see Section 3.5.
- The pragma FUNC_ALWAYS_INLINE (Section 5.11.12) and the equivalent `always_inline` attribute (Section 5.17.2) force a function to be inlined (where it is legal to do so) unless --opt_level=off. That is, the pragma FUNC_ALWAYS_INLINE forces function inlining even if the function is not declared as inline and the --opt_level=0 or --opt_level=1.
- The FORCEINLINE pragma (Section 5.11.10) forces functions to be inlined in the annotated statement. That is, it has no effect on those functions in general, only on function calls in a single statement. The FORCEINLINE_RECURSIVE pragma forces inlining not only of calls visible in the statement, but also in the inlined bodies of calls from that statement.
- The --disable_inlining option prevents any inlining. The pragma FUNC_CANNOT_INLINE prevents a function from being inlined. The NOINLINE pragma prevents calls within a single statement from being inlined. (NOINLINE is the inverse of the FORCEINLINE pragma.)

Note

Function Inlining Can Greatly Increase Code Size: Function inlining increases code size, especially inlining a function that is called in a number of places. Function inlining is optimal for functions that are called only from a small number of places and for small functions.

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

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

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

2.11.1 Inlining Intrinsic Operators

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

Intrinsics are inlined whether or not you use the optimizer. For details about intrinsics, and a list of the intrinsics, see Section 5.14. In addition to those listed, abs and memcpy are implemented as intrinsics.

2.11.2 Inlining Restrictions

The compiler makes decisions about which functions to inline based on the factors mentioned in Section 2.11. In addition, there are several restrictions that can disqualify a function from being inlined by automatic inlining or inline keyword-based inlining.

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

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

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

- Has a variable-length argument list
- Never returns
- Is a recursive or non-leaf function that exceeds the depth limit
- Is not declared inline and contains an asm() statement that is not a comment
- Is an interrupt function
- Is the main() function
- Is not declared inline and will require too much stack space for local array or structure variables
- Contains a volatile local variable or argument
- Is a C++ function that contains a catch
- Is not defined in the current compilation unit and -O4 optimization is not used

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

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

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

2.12 Using Interlist

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

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

```
armcl --c_src_interlist function
```

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

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

For information about using the interlist feature with the optimizer, see Section 3.11. Using the --c_src_interlist option can cause performance and/or code size degradation.

The following example shows a typical interlisted assembly file.

```
_main:
        STMFD     SP!, {LR}
;-------------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;-------------------------------------------------------------------------
        ADR       A1, SL1
        BL        _printf
;-------------------------------------------------------------------------
;   6 | return 0;
;-------------------------------------------------------------------------
        MOV       A1, #0
        LDMFD     SP!, {PC}
```

2.13 Controlling Application Binary Interface

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

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

The ARM compiler now supports only the Embedded Application Binary Interface (EABI) ABI, which uses the ELF object format and the DWARF debug format. If you want support for the legacy TI_ARM9_ABI and TIARM ABIs, please use the ARM v5.2 Code Generation Tools and refer to SPNU151J and SPNU118J for documentation.

An industry consortium founded by ARM Ltd defined a standard ABI for binary code intended for the ARM architecture. This ABI is called the Application Binary Interface (ABI) for the ARM Architecture Version 2 (ARM ABIv2). This ABI is also referred to as Embedded Application Binary Interface (EABI). The terms ABIv2 and EABI can be used interchangeably.

For more details on the ABI, see Section 5.13.

2.14 VFP Support

The compiler includes support for generating vector floating-point (VFP) co-processor instructions through the --float_support=*vfp* option. The VFP co-processor is available in many variants of ARM11 and higher. The valid *vfp* entries are:

vfpv2 Allows generation of floating point instructions for ARM9E.

vfpv3 Allows generation of floating point instructions for Cortex-A8.

vfpv3d16 Allows generation of floating point instructions for Cortex-R4.

fpv4spd16 Allows generation of floating point instructions for Cortex-M4.

none Disables hardware floating point support. Specifies that the compiler implements floating point operations in software.

Using the --silicon_version=7M4 command-line option automatically sets the --float_support=fpv4spd16 option. To disable hardware floating point support, use the --float_support=none option.

This is the current support for VFP:

- You must link any VFP compiled code with a separate version of the run-time support library. See Section 7.1.9 for information on library-naming conventions.
- The compiler follows the VFP argument passing and returning calling convention for qualified VFP arguments.
- Object files that *do not contain* any functions with floating point arguments or return values can be linked with both VFP and non-VFP files.
- Object files that *do contain* functions with floating point arguments or return values can only be linked with objects that were compiled with matching VFP support.
- All hand-coded VFP assembly must follow VFP calling conventions and EABI conventions to correctly compile and link. In addition to these, the appropriate VFP build attributes for EABI must be correctly set.
- The compile-time predefined macro __TI_VFP_SUPPORT__ can be used for conditionally compiling/assembling user code. VFP-specific user code can use this macro to ensure that the conditionally included code is compiled only when VFP is enabled.

Refer to the ARM architecture manual for more details on the VFPv3 and VFPv3D16 architectures and ISAs. Refer to the ARM AAPCS and EABI documents for more details on VFP calling conventions and build attributes.

2.15 Enabling Entry Hook and Exit Hook Functions

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

--entry_hook[=*name*]	Enables entry hooks. If specified, the hook function is called *name*. Otherwise, the default entry hook function name is __entry_hook.
--entry_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 5.11.21 for information about the NO_HOOKS pragma.

This page intentionally left blank.

Chapter 3

Optimizing Your Code

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

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

3.1 Invoking Optimization 54
3.2 Controlling Code Size Versus Speed 55
3.3 Performing File-Level Optimization (--opt_level=3 option) 55
3.4 Program-Level Optimization (--program_level_compile and --opt_level=3 options) 56
3.5 Automatic Inline Expansion (--auto_inline Option) 58
3.6 Link-Time Optimization (--opt_level=4 Option) 59
3.7 Using Feedback Directed Optimization 60
3.8 Using Profile Information to Analyze Code Coverage 63
3.9 Accessing Aliased Variables in Optimized Code 65
3.10 Use Caution With asm Statements in Optimized Code 65
3.11 Using the Interlist Feature With Optimization 65
3.12 Debugging and Profiling Optimized Code 66
3.13 What Kind of Optimization Is Being Performed? 67

3.1 Invoking Optimization

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

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

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

The easiest way to invoke optimization is to use the compiler program, specifying the --opt_level=*n* option on the compiler command line. You can use -O*n* as an alias for the --opt_level option. The *n* denotes the level of optimization (0, 1, 2, 3, and 4), which controls the type and degree of optimization.

- **--opt_level=off** or **-Ooff**
  - Performs no optimization
- **--opt_level=0** or **-O0**
  - Performs control-flow-graph simplification (Section 3.13.3)
  - Performs loop rotation (Section 3.13.10)
  - Eliminates unused code
  - Simplifies expressions and statements (Section 3.13.5)
  - Expands calls to functions declared as inline (Section 3.13.6)
- **--opt_level=1** or **-O1** Performs all --opt_level=0 (-O0) optimizations, plus:
  - Performs local copy/constant propagation (Section 3.13.4)
  - Removes unused assignments (Section 3.13.4)
  - Eliminates local common expressions
- **--opt_level=2** or **-O2** Performs all --opt_level=1 (-O1) optimizations, plus:
  - Performs loop optimizations
  - Eliminates global common subexpressions (Section 3.13.4)
  - Eliminates global unused assignments (Section 3.13.4)
  - Performs loop unrolling (Section 5.11.31)
- **--opt_level=3** or **-O3** Performs all --opt_level=2 (-O2) optimizations, plus:
  - Removes all functions that are never called (Section 3.4)
  - Simplifies functions with return values that are never used (Section 3.4)
  - Inlines calls to small functions (Section 2.11 and Section 3.5)
  - Reorders function declarations; the called functions attributes are known when the caller is optimized
  - Propagates arguments into function bodies when all calls pass the same value in the same argument position
  - Identifies file-level variable characteristics (Section 3.4)
  - Performs other optimizations (Section 3.3 and Section 3.4)
- **--opt_level=4** or **-O4**
  - Performs link-time optimization. (Section 3.6)

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

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

3.2 Controlling Code Size Versus Speed

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

- --opt_for_speed=0

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

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

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

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

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

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

If you specify the --opt_for_speed option without a parameter, the default setting is --opt_for_speed=4. If you do not specify the --opt_for_speed option, the default setting is 1

The best performance for caching devices has been observed with --opt_for_speed set to level 1 or 2.

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

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

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

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

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

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

Table 3-2. Selecting a Level for the --gen_opt_info Option

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

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

You can specify program-level optimization by using the --program_level_compile option with the --opt_level=3 option (aliased as -O3). (If you use --opt_level=4 (-O4), the --program_level_compile option cannot be used, because link-time optimization provides the same optimization opportunities as program level optimization.)

With program-level optimization, all of your source files are compiled into one intermediate file called a *module*. The module moves to the optimization and code generation passes of the compiler. Because the compiler can see the entire program, it performs several optimizations that are rarely applied during file-level optimization:

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

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

To see which program-level optimizations the compiler is applying, use the --gen_opt_level=2 option to generate an information file. See Section 3.3.1 for more information.

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

Note

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

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

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

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

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

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

In certain circumstances, the compiler reverts to a different --call_assumptions level from the one you specified, or it might disable program-level optimization altogether. Table 3-4 lists the combinations of --call_assumptions levels and conditions that cause the compiler to revert to other --call_assumptions levels.

Table 3-4. Special Considerations When Using the --call_assumptions Option

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

In some situations when you use --program_level_compile and --opt_level=3, you *must* use a --call_assumptions option or the FUNC_EXT_CALLED pragma. See Section 3.4.2 for information about these situations.

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

If you have any assembly functions in your program, you need to exercise caution when using the --program_level_compile option. The compiler recognizes only the C/C++ source code and not any assembly code that might be present. Because the compiler does not recognize the assembly code calls and variable modifications to C/C++ functions, the --program_level_compile option optimizes out those C/C++ functions. To keep these functions, place the FUNC_EXT_CALLED pragma (see Section 5.11.14) before any declaration or reference to a function that you want to keep.

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

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

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

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

  Solution: Compile with --program_level_compile --opt_level=3 --call_assumptions=2 to tell the compiler that outside functions do not call C/C++ functions or modify C/C++ variables.

  If you compile with the --program_level_compile --opt_level=3 options only, the compiler reverts from the default optimization level (--call_assumptions=2) to --call_assumptions=0. The compiler uses --call_assumptions=0, because it presumes that the calls to the assembly language functions that have a definition in C/C++ may call other C/C++ functions or modify C/C++ variables.
- **Situation:** Your application consists of C/C++ source code that calls assembly functions. The assembly language functions do not call C/C++ functions, but they modify C/C++ variables.

  Solution: Try both of these solutions and choose the one that works best with your code:
  - Compile with --program_level_compile --opt_level=3 --call_assumptions=1.
  - Add the volatile keyword to those variables that may be modified by the assembly functions and compile with --program_level_compile --opt_level=3 --call_assumptions=2.
- **Situation:** Your application consists of C/C++ source code and assembly source code. The assembly functions are interrupt service routines that call C/C++ functions; the C/C++ functions that the assembly functions call are never called from C/C++. These C/C++ functions act like main: they function as entry points into C/C++.

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

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

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

3.5 Automatic Inline Expansion (--auto_inline Option)

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

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

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

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

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

- The code size limit is smaller when compiling for code size rather than performance. The --auto_inline option overrides this size limit.
- At --opt_level=3, the compiler automatically inlines small functions.
- At --opt_level=4, the compiler auto-inlines aggressively if compiling for performance.

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

Note

Some Functions Cannot Be Inlined: For a call-site to be considered for inlining, it must be legal to inline the function and inlining must not be disabled in some way. See the inlining restrictions in Section 2.11.2.

Note

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

Note

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

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

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

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

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

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

3.6.1 Option Handling

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

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

3.6.2 Incompatible Types

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

3.7 Using Feedback Directed Optimization

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

3.7.1 Feedback Directed Optimization

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

3.7.1.1 Phase 1 -- Collect Program Profile Information

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

The instrumented application program is executed on the target using representative input data sets. The input data sets should correlate closely with the way the program is expected to be used in the end product environment. When the program completes, a run-time-support function writes the collected information into a profile data file called a PDAT file. Multiple executions of the program using different input data sets can be performed and in such cases, the run-time-support function appends the collected information into the PDAT file. The resulting PDAT file is post-processed using a tool called the Profile Data Decoder or armpdd. The armpdd tool consolidates multiple data sets and formats the data into a feedback file (PRF file, see Section 3.7.2) for consumption by phase 2 of feedback directed optimization.

3.7.1.2 Phase 2 -- Use Application Profile Information for Optimization

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

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

3.7.1.3 Generating and Using Profile Information

There are two options that control feedback directed optimization:

--gen_profile_info tells the compiler to add instrumentation code to collect profile information. When the program executes the run-time-support exit() function, the profile data is written to a PDAT file. This option applies to all the C/C++ source files being compiled on the command-line.
If the environment variable TI_PROFDATA on the host is set, the data is written into the specified file. Otherwise, it uses the default filename: pprofout.pdat. The full pathname of the PDAT file (including the directory name) can be specified using the TI_PROFDATA host environment variable.
By default, the RTS profile data output routine uses the C I/O mechanism to write data to the PDAT file. You can install a device handler for the PPHNDL device to re-direct the profile data to a custom device driver routine. For example, this could be used to send the profile data to a device that does not use a file system.
Feedback directed optimization requires you to turn on at least some debug information when using the --gen_profile_info option. This enables the compiler to output debug information that allows armpdd to correlate compiled functions and their associated profile data.

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

3.7.1.4 Example Use of Feedback Directed Optimization

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

1. Generate profile information.

```
armcl --opt_level=2 --gen_profile_info foo.c --run_linker --output_file=foo.out
     --library=lnk.cmd --library=rtsv4_A_be_eabi.lib
```

2. Execute the application.

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

3. Process the profile data.

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

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

4. Re-compile using the profile feedback file.

```
armcl --opt_level=2 --use_profile_info=pprofout.prf foo.c --run_linker
     --output_file=foo.out --library=lnk.cmd --library=rtsv4_A_be_eabi.lib
```

3.7.1.5 The .ppdata Section

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

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

3.7.1.6 Feedback Directed Optimization and Code Size Tune

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

3.7.1.7 Instrumented Program Execution Overhead

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

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

3.7.1.8 Invalid Profile Data

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

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

3.7.2 Profile Data Decoder

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

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

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

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

Note

Profile Data Decoder Requirements: Compile applications with at least DWARF debug support to enable feedback-directed optimization. When compiling for feedback-directed optimization, the armpdd tool relies on basic debug information about each function to generate the formatted .prf file.

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

3.7.3 Feedback Directed Optimization API

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

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

3.7.4 Feedback Directed Optimization Summary

Options

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

Host Environment Variables

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

API

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

Files Created

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

3.8 Using Profile Information to Analyze Code Coverage

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

3.8.1 Code Coverage

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

3.8.1.1 Phase1 -- Collect Program Profile Information

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

The instrumented application program is executed on the target using representative input data sets. The input data sets should correlate closely with the way the program is expected to be used in the end product environment. When the program completes, a run-time-support function writes the collected information into a profile data file called a PDAT file. Multiple executions of the program using different input data sets can be performed and in such cases, the run-time-support function appends the collected information into the PDAT file. The resulting PDAT file is post-processed using a tool called the Profile Data Decoder or armpdd. The armpdd tool consolidates multiple data sets and formats the data into a feedback file (PRF file, see Section 3.7.2) for consumption by phase 2 of feedback directed optimization.

3.8.1.2 Phase 2 -- Generate Code Coverage Reports

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

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

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

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

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

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

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

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

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

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

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

3.8.2 Related Features and Capabilities

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

3.8.2.1 Path Profiler

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

armcl --gen_profile ... *file.c*
armcl --use_profile ... *file.c*

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

3.8.2.2 Analysis Options

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

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

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

3.8.2.3 Environment Variables

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

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

3.9 Accessing Aliased Variables in Optimized Code

Aliasing occurs when a single object can be accessed in more than one way, such as when two pointers point to the same object or when a pointer points to a named object. Aliasing can disrupt optimization because any indirect reference can refer to another object. The optimizer analyzes the code to determine where aliasing can and cannot occur, then optimizes as much as possible while still preserving the correctness of the program. The optimizer behaves conservatively. If there is a chance that two pointers are pointing to the same object, then the optimizer assumes that the pointers do point to the same object.

The compiler assumes that if the address of a local variable is passed to a function, the function changes the local variable by writing through the pointer. This makes the local variable's address unavailable for use elsewhere after returning. For example, the called function cannot assign the local variable's address to a global variable or return the local variable's address. In cases where this assumption is invalid, use the --aliased_variables compiler option to force the compiler to assume worst-case aliasing. In worst-case aliasing, any indirect reference can refer to such a variable.

3.10 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.

3.11 Using the Interlist Feature With Optimization

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

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

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

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

Note

Impact on Performance and Code Size: The --c_src_interlist option can have a negative effect on performance and code size.

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

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

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

The assembly file contains compiler comments interlisted with assembly code.

```
_main:
        STMFD     SP!, {LR}
;** 5-----------------------    printf("Hello, world\n");
        ADR       A1, SL1
        BL        _printf
;** 6-----------------------    return 0;
        MOV       A1, #0
        LDMFD     SP!, {PC}
```

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

```
_main:
        STMFD     SP!, {LR}
;** 5-----------------------    printf("Hello, world\n");
;------------------------------------------------------------------------------
;   5 | printf("Hello, world\n");
;------------------------------------------------------------------------------
        ADR       A1, SL1
        BL        _printf
;** 6-----------------------    return 0;
;------------------------------------------------------------------------------
;   6 | return 0;
;------------------------------------------------------------------------------
        MOV       A1, #0
        LDMFD     SP!, {PC}
```

3.12 Debugging and Profiling Optimized Code

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

The default optimization level depends on the use of the --symdebug:dwarf (-g) option. If --symdebug:dwarf is specified, the default optimization level is off. Otherwise the default optimization level is 3.

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

3.12.1 Profiling Optimized Code

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

3.13 What Kind of Optimization Is Being Performed?

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

Optimization	See
Cost-based register allocation	Section 3.13.1
Alias disambiguation	Section 3.13.2
Branch optimizations and control-flow simplification	Section 3.13.3
Data flow optimizations • Copy propagation • Common subexpression elimination • Redundant assignment elimination	Section 3.13.4
Expression simplification	Section 3.13.5
Inline expansion of functions	Section 3.13.6
Function symbol aliasing	Section 3.13.7
Induction variables and strength reduction	Section 3.13.8
Loop-invariant code motion	Section 3.13.9
Loop rotation	Section 3.13.10
Instruction scheduling	Section 3.13.11

ARM-Specific Optimization	See
Tail merging	Section 3.13.12
Autoincrement addressing	Section 3.13.13
Block conditionalizing	Section 3.13.14
Epilog inlining	Section 3.13.15
Removing comparisons to zero	Section 3.13.16
Integer division with constant divisor	Section 3.13.17
Branch chaining	Section 3.13.18

3.13.1 Cost-Based Register Allocation

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

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

3.13.2 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.

3.13.3 Branch Optimizations and Control-Flow Simplification

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

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

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

3.13.4 Data Flow Optimizations

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

- **Copy propagation.** Following an assignment to a variable, the compiler replaces references to the variable with its value. The value can be another variable, a constant, or a common subexpression. This can result in increased opportunities for constant folding, common subexpression elimination, or even total elimination of the variable. This type of optimization is enabled by the --opt_level=1 and higher optimization settings.
- **Common subexpression elimination.** When two or more expressions produce the same value, the compiler computes the value once, saves it, and reuses it. This type of optimization is enabled by the --opt_level=2 and higher optimization settings.
- **Redundant assignment elimination.** Often, copy propagation and common subexpression elimination optimizations result in unnecessary assignments to variables (variables with no subsequent reference before another assignment or before the end of the function). The compiler removes these dead assignments. This type of optimization is enabled by the --opt_level=1 for local assignments and --opt_level=2 for global assignments.

3.13.5 Expression Simplification

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

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

3.13.6 Inline Expansion of Functions

The compiler replaces calls to small functions with inline code, saving the overhead associated with a function call as well as providing increased opportunities to apply other optimizations. For information about interactions between command-line options, pragmas, and keywords that affect inlining, see Section 2.11.

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

3.13.7 Function Symbol Aliasing

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

For example, consider the following:

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

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

For information about using the GCC function attribute syntax to declare function aliases, see Section 5.17.2

3.13.8 Induction Variables and Strength Reduction

Induction variables are variables whose value within a loop is directly related to the number of executions of the loop. Array indices and control variables for loops are often induction variables.

Strength reduction is the process of replacing inefficient expressions involving induction variables with more efficient expressions. For example, code that indexes into a sequence of array elements is replaced with code that increments a pointer through the array.

Induction variable analysis and strength reduction together often remove all references to your loop-control variable, allowing its elimination.

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

3.13.9 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.

3.13.10 Loop Rotation

The compiler evaluates loop conditionals at the bottom of loops, saving an extra branch out of the loop. In many cases, the initial entry conditional check and the branch are optimized out.

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

3.13.11 Instruction Scheduling

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

3.13.12 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.

3.13.13 Autoincrement Addressing

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

3.13.14 Block Conditionalizing

Because all 32-bit instructions can be conditional, branches can be removed by conditionalizing instructions.

In Example 3-1, the branch around the add and the branch around the subtract are removed by simply conditionalizing the add and the subtract.

Example 3-1. Block Conditionalizing C Source

```
int main(int a)
{
   if (a < 0)
      a = a-3;
   else
      a = a*3;
   return ++a;
}
```

Example 3-2. C/C++ Compiler Output for Example 3-1

```
;*****************************************************
;* FUNCTION DEF: _main                               *
;*****************************************************
_main:
          CMP       A1, #0
          ADDPL     A1, A1, A1, LSL #1
          SUBMI     A1, A1, #3
          ADD       A1, A1, #1
          BX        LR
```

3.13.15 Epilog Inlining

If the epilog of a function is a single instruction, that instruction replaces all branches to the epilog. This increases execution speed by removing the branch.

3.13.16 Removing Comparisons to Zero

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

3.13.17 Integer Division With Constant Divisor

The optimizer attempts to rewrite integer divide operations with constant divisors. The integer divides are rewritten as a multiply with the reciprocal of the divisor. This occurs at optimization level 2 (--opt_level=2 or -O2) and higher. You must also compile with the --opt_for_speed option, which selects compile for speed.

3.13.18 Branch Chaining

Branching to branches that jump to the desired target is called branch chaining. Branch chaining is supported in 16-BIS mode only. Consider this code sequence:

```
LAB1:   BR  L10
        ....
LAB2:   BR  L10
        ....
L10:
```

If L10 is far away from LAB1 (large offset), the assembler converts BR into a sequence of branch around and unconditional branches, resulting in a sequence of two instructions that are either four or six bytes long. Instead, if the branch at LAB1 can jump to LAB2, and LAB2 is close enough that BR can be replaced by a single short branch instruction, the resulting code is smaller as the BR in LAB1 would be converted into one instruction that is two bytes long. LAB2 can in turn jump to another branch if L10 is too far away from LAB2. Thus, branch chaining can be extended to arbitrary depths.

When you compile in thumb mode (--code_state=16) and for code size (--opt_for_speed is not used), the compiler generates two psuedo instructions:

- BTcc instead of BRcc. The format is **BRcc** *target*, #[*depth*].

  The *#depth* is an optional argument. If depth is not specified, it is set to the default branch chaining depth. If specified, the chaining depth for this branch instruction is set to #depth. The assembler issues a warning if #depth is less than zero and sets the branch chaining depth for this instruction to zero.
- BQcc instead of Bcc. The format is **BQcc** *target* , #[*depth*].

  The *#depth* is the same as for the BTcc psuedo instruction.

The BT pseudo instruction replaces the BR (pseudo branch) instruction. Similarly, BQ replaces B. The assembler performs branch chain optimizations for these instructions, if branch chaining is enabled. The assembler replaces the BT and BQ jump targets with the offset to the branch to which these instructions jump.

The default branch chaining depth is 10. This limit is designed to prevent longer branch chains from impeding performance.

You can the BT and BQ instructions in assembly language programs to enable the assembler to perform branch chaining. You can control the branch chaining depth for each instruction by specifying the (optional) *#depth* argument. You must use the BR and B instructions to prevent branch chaining for any BT or BQ branches.

This page intentionally left blank.

Chapter 4

Linking C/C++ Code

The C/C++ Code Generation Tools provide two methods for linking your programs:

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

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

4.1 Invoking the Linker Through the Compiler (-z Option) 74
4.2 Linker Code Optimizations 76
4.3 Controlling the Linking Process 77

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:

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

armcl --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 armcl --run_linker without listing any C/C++ files to be compiled on the command line, you *must* use **--rom_model** or **--ram_model** on the command line or in the linker command file. The --rom_model option uses automatic variable initialization at run time; the --ram_model option uses variable initialization at load time. See Section 4.3.5 for details about using the --rom_model and --ram_model options. If you fail to specify the ROM or RAM model, you will see a linker warning that says: `warning: no suitable entry-point found; setting to 0`
filenames	Names of object files, linker command files, or archive libraries. The default extensions for input files are .c.obj (for C source files) and .cpp.obj (for C++ source files). Any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is *a.out*, unless you use the --output_file option.
options	Options affect how the linker handles your object files. Linker options can only appear after the **--run_linker** option on the command line, but otherwise may be in any order. (Options are discussed in detail in the *ARM Assembly Language Tools User's Guide*.)
--output_file= *name.out*	Names the output file.
--library= *library*	Identifies the appropriate archive library containing C/C++ run-time-support and floating-point math functions, or linker command files. If you are linking C/C++ code, you must use a run-time-support library. You can use the libraries included with the compiler, or you can create your own run-time-support library. If you have specified a run-time-support library in a linker command file, you do not need this parameter. The --library option's short form is -l.
lnk.cmd	Contains options, filenames, directives, or commands for the linker.

Note

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

When you specify a library as linker input, the linker includes and links only those library members that resolve undefined references. The linker uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS directives in the linker command file to customize the allocation process. For information, see the *ARM Assembly Language Tools User's Guide*.

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

```
armcl --run_linker --rom_model prog1 prog2 prog3 --output_file=prog.out
     --library=rtsv4_A_be_eabi.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:

armcl *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:

```
armcl prog1.c prog2.c prog3.c --run_linker --rom_model --output_file=prog.out
     --library=rtsv4_A_be_eabi.lib
```

When you use armcl --run_linker *after* listing at least one C/C++ file to be compiled on the same command line, by default the **--rom_model** is used for automatic variable initialization at run time. See Section 4.3.5 for details about using the --rom_model and --ram_model options.

Note

Order of Processing Arguments in the Linker: The order in which the linker processes arguments is important. The compiler passes arguments to the linker in the following order:

1. Object filenames from the command line
2. Arguments following the --run_linker option on the command line
3. Arguments following the --run_linker option from the TI_ARM_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 TI_ARM_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 techniques 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:

   ```
   armcl 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:

   ```
   armcl --run_linker file1.c.obj file2.c.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:

   ```
   armcl 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:

   ```
   armcl 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:

   ```
   armcl --run_linker file1.c.obj file2.c.obj
   ```

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

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

Note

Dead Functions Feedback: The format of the feedback file generated with --generate_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 Aggregate Data Subsections (--gen_data_subsections Compiler Option)

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

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

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

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 *ARM Assembly Language Tools User's Guide*.

4.3.1 Including the Run-Time-Support Library

You must link all C/C++ programs with a run-time-support library. The library contains standard C/C++ functions as well as functions used by the compiler to manage the C/C++ environment. The following sections describe two methods for including the run-time-support library.

4.3.1.1 Automatic Run-Time-Support Library Selection

The linker assumes you are using the C and C++ conventions if either the --rom_model or --ram_model linker option is specified, or if at least one C/C++ file to compile is listed on the command line. See Section 4.3.5 for details about using the --rom_model and --ram_model options.

If the linker assumes you are using the C and C++ conventions and the entry point for the program (normally c_int00) is not resolved by any specified object file or library, the linker attempts to automatically include the most compatible run-time-support library for your program. The run-time-support library chosen by the compiler is searched after any other libraries specified with the --library option on the command line or in the linker command file. If libc.a is explicitly used, the appropriate run-time-support library is included in the search order where libc.a is specified.

You can disable the automatic selection of a run-time-support library by using the --disable_auto_rts option.

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

Example 4-1. Using the --issue_remarks Option

```
armcl --code_state=16 --issue_remarks main.c --run_linker --rom_model
<Linking>
remark: linking in "libc.a"
remark: linking in "rtsv4_A_be_eabi.lib" in place of "libc.a"
```

4.3.1.2 Manual Run-Time-Support Library Selection

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

armcl --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. Switch to user mode and sets up the user mode stack
2. Set up status and configuration registers
3. Set up the stack
4. Process special binit copy table, if present.
5. Process the run-time initialization table to autoinitialize global variables (when using the --rom_model option)
6. Call all global constructors
7. Call the main() function
8. Call exit() when main() returns

Note

The _c_int00 Symbol: If you use the --ram_model or --rom_model link option, _c_int00 is automatically defined as the entry point for the program. If your command line does not list any C/C++ files to compile and does not specify either the --ram_model or --rom_model link option, the linker does not know whether or not to use the C/C++ conventions, and you will receive a linker warning that says "warning: no suitable entry-point found; setting to 0". See Section 4.3.5 for details about using the --rom_model and --ram_model options.

4.3.3 Initialization of Cinit and Watchdog Timer Hold

You can use the --cinit_hold_wdt option to specify whether the watchdog timer should be held (on) or not held (off) during cinit auto-initialization. Setting this option causes an RTS auto-initialization routine to be linked in with the program to handle the desired watchdog timer behavior.

4.3.4 Global Object Constructors

Global C++ variables that have constructors and destructors require their constructors to be called during program initialization and their destructors to be called during program termination. The C++ compiler produces a table of constructors to be called at startup.

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

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

Section 6.10.3.6 discusses the format of the global constructor table for EABI mode.

4.3.5 Specifying the Type of Global Variable Initialization

The C/C++ compiler produces data tables for initializing global variables. Section 6.10.3.4 discusses the format of these initialization tables. The initialization tables are used in one of the following ways:

- Global variables are initialized at *run time*. Use the --rom_model linker option (see Section 6.10.3.3).
- Global variables are initialized at *load time*. Use the --ram_model linker option (see Section 6.10.3.5).

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

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

If you use a single command line to both compile and link, the --rom_model option is the default. If used, the --rom_model or --ram_model option must follow the --run_linker option (see Section 4.1).

For details on linking conventions for EABI with --rom_model and --ram_model, see Section 6.10.3.3 and Section 6.10.3.5, respectively.

Note

Boot Loader: A loader is not included as part of the C/C++ compiler tools. You can use the ARM simulator or emulator with the source debugger as a loader. See the "Program Loading and Running" chapter of the *ARM Assembly Language Tools User's Guide* for more about boot loading.

4.3.6 Specifying Where to Allocate Sections in Memory

The compiler produces relocatable blocks of code and data. These blocks, called *sections*, are allocated in memory in a variety of ways to conform to a variety of system configurations. See Section 6.1.1 for a complete description of how the compiler uses these sections.

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

Table 4-1. Initialized Sections Created by the Compiler

Name	Contents
.binit	Boot time copy tables (See the *Assembly Language Tools User's Guide* for information on BINIT in linker command files.)
.cinit	Tables for explicitly initialized global and static variables.
.const	Global and static const variables that are explicitly initialized.
.data	Global and static non-const variables that are explicitly initialized.
.init_array	Table of constructors to be called at startup.
.ovly	Copy tables other than boot time (.binit) copy tables. Read-only data.
.text	Executable code and constants. Also contains string literals and switch tables. See Section 6.1.1 for exceptions.
.TI.crctab	Generated CRC checking tables. Read-only data.

Table 4-2. Uninitialized Sections Created by the Compiler

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

When you link your program, you must specify where to allocate the sections in memory. In general, initialized sections are linked into ROM or RAM; uninitialized sections are linked into RAM.

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

4.3.7 A Sample Linker Command File

Linker Command File shows a typical linker command file that links a 32-bit C program. The command file in this example is named lnk32.cmd and lists several link options:

--rom_model	Tells the linker to use autoinitialization at run time
--stack_size	Tells the linker to set the C stack size at 0x8000 bytes
--heap_size	Tells the linker to set the heap size to 0x2000 bytes

To link the program, use the following syntax:

armcl --run_linker *object_file*(s) **--output_file** *outfile* **--map_file** *mapfile* **lnk32.cmd**

Linker Command File

```
--rom_model                                  /* LINK USING C CONVENTIONS     */
--stack_size=0x8000                          /* SOFTWARE STACK SIZE          */
--heap_size=0x2000                           /* HEAP AREA SIZE               */
/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY
{
    P_MEM    : org = 0x00000000   len = 0x00030000   /* PROGRAM MEMORY (ROM) */
    D_MEM    : org = 0x00030000   len = 0x00050000   /* DATA MEMORY    (RAM) */
}
/* SPECIFY THE SECTIONS ALLOCATION INTO MEMORY */
SECTIONS
{
    .intvecs : {} > 0x0                  /* INTERRUPT VECTORS                */
    .bss     : {} > D_MEM                /* GLOBAL & STATIC VARS             */
    .sysmem  : {} > D_MEM                /* DYNAMIC MEMORY ALLOCATION AREA   */
    .stack   : {} > D_MEM                /* SOFTWARE SYSTEM STACK            */
    .text    : {} > P_MEM                /* CODE                             */
    .cinit   : {} > P_MEM                /* INITIALIZATION TABLES            */
    .const   : {} > P_MEM                /* CONSTANT DATA                    */
    .pinit   : {} > P_MEM                /* TEMPLATE INITIALIZATION TABLES   */
```

Chapter 5

C/C++ Language Implementation

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

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

5.1 Characteristics of ARM C 82
5.2 Characteristics of ARM C++ 87
5.3 Using MISRA C 2004 88
5.4 Using the ULP Advisor 89
5.5 Data Types 90
5.6 File Encodings and Character Sets 92
5.7 Keywords 92
5.8 C++ Exception Handling 95
5.9 Register Variables and Parameters 95
5.10 The __asm Statement 97
5.11 Pragma Directives 98
5.12 The _Pragma Operator 117
5.13 Application Binary Interface 118
5.14 ARM Instruction Intrinsics 118
5.15 Object File Symbol Naming Conventions (Linknames) 127
5.16 Changing the ANSI/ISO C/C++ Language Mode 128
5.17 GNU , Clang, and ACLE Language Extensions 130
5.18 AUTOSAR 136
5.19 Compiler Limits 136

5.1 Characteristics of ARM C

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

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

The C language is also described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R). The compiler can also accept many of the language extensions found in the GNU C compiler (see Section 5.17).

The compiler supports some features of C99 and C11 in the default relaxed ANSI mode with C89 support. It supports all language features of C99 in C99 mode and all language features of C11 in C11 mode. See Section 5.16.

The atomic operations in C11 are supported in the relaxed ANSI mode (on by default) and in C11 mode as follows:

- On ARM V7A8 (Cortex-A8), ARM V7M3 (Cortex-M3), ARM V7M4 (Cortex-M4), ARM V7R4 (Cortex-R4), and ARM V7R5 (Cortex-R5)), atomic operations are implemented using processor-supported exclusive access instructions.
- On ARM V6M0 (Cortex-M0), atomic operations are implemented by disabling interrupts across the operation.
- On ARM V4 (ARM7), ARM V5e (ARM9E), and ARM V6 (ARM11), atomic operations are not supported.

In addition, the compiler supports many of the features described in the ARM C Language Extensions (ACLE) specification. These features are applicable for the Cortex-M and Cortex-R processor variants. ACLE support affects the pre-defined macros (Table 2-31), function attributes (Section 5.17.2), and intrinsics (Section 5.14) you may use in C/C++ code. These features are implemented in order to support the development of source code that can be compiled using ACLE-compliant compilers from multiple vendors for a variety of ARM processors.

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

Unsupported features of the C library are:

- The run-time library has minimal support for wide characters. The type wchar_t is implemented as unsigned short (16 bits), but can be an int if you set the --wchar_t=32 option. The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h>, but does not include all the functions specified in the standard. See Section 5.6 for information about extended and multibyte character sets.
- The run-time library includes the header file <locale.h>, but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hard-coded to the behavior of the C locale, and attempting to install a different locale by way of a call to setlocale() will return NULL.
- Some run-time functions and features in the C99/C11 specifications are not supported. See Section 5.16.

5.1.1 Implementation-Defined Behavior

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

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

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

J.3.1 Translation

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

J.3.2 Environment

- The compiler does not support multibyte characters in identifiers, string literals, or character constants. There is no mapping from multibyte characters to the source character set. However, the compiler accepts multibyte characters in comments. See Section 5.6 for details (5.1.1.2)
- The name of the function called at program startup is "main". (5.1.2.1)
- Program termination does not affect the environment; there is no way to return an exit code to the environment. By default, the program is known to have halted when execution reaches the special C$$EXIT label. (5.1.2.1)
- In relaxed ANSI mode, the compiler accepts "void main(void)" and "void main(int argc, char *argv[])" as alternate definitions of main. The alternate definitions are rejected in strict ANSI mode. (5.1.2.2.1)
- If space is provided for program arguments at link time with the --args option and the program is run under a system that can populate the .args section (such as CCS), argv[0] will contain the filename of the executable, argv[1] through argv[argc-1] will contain the command-line arguments to the program, and argv[argc] will be NULL. Otherwise, the value of argv and argc are undefined. (5.1.2.2.1)
- Interactive devices include stdin, stdout, and stderr (when attached to a system that honors CIO requests). Interactive devices are not limited to those output locations; the program may access hardware peripherals that interact with the external state. (5.1.2.3)
- Signals are not supported. The function signal is not supported. (7.14, 7.14.1.1)
- The library function getenv is implemented through the CIO interface. If the program is run under a system that supports CIO, the system performs getenv calls on the host system and passes the result back to the program. Otherwise the operation of getenv is undefined. No method of changing the environment from inside the target program is provided. (7.20.4.5)
- The system function is not supported. (7.20.4.6)

J.3.3. Identifiers

- The compiler does not support multibyte characters in identifiers. See Section 5.6 for details. (6.4.2)
- The number of significant initial characters in an identifier is unlimited. (5.2.4.1, 6.4.2)

J.3.4 Characters

- The number of bits in a byte (CHAR_BIT) is 8. See Section 5.5 for details about data types. (3.6)
- The execution character set is the same as the basic execution character set: plain ASCII. Characters in the ISO 8859 extended character set are also supported. (5.2.1)
- The values produced for the standard alphabetic escape sequences are as follows. (5.2.2):

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

- The value of a char object into which any character other than a member of the basic execution character set has been stored is the ASCII value of that character. (6.2.5)
- Plain char is identical to unsigned char, but can be changed to signed char with the --plain_char=signed option. (6.2.5, 6.3.1.1)
- The source character set and execution character set are both plain ASCII, so the mapping between them is one-to-one. The compiler accepts multibyte characters in comments. See Section 5.6 for details. (6.4.4.4, 5.1.1.2)
- The compiler currently supports only one locale, "C". (6.4.4.4)
- The compiler currently supports only one locale, "C". (6.4.5)

J.3.5 Integers

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

J.3.6 Floating point

- The accuracy of floating-point operations (+ - * /) is bit-exact. The accuracy of library functions that return floating-point results is not specified. (5.2.4.2.2)
- The compiler does not provide non-standard values for FLT_ROUNDS. (5.2.4.2.2)
- The compiler does not provide non-standard negative values of FLT_EVAL_METHOD. (5.2.4.2.2)
- The rounding direction when an integer is converted to a floating-point number is IEEE-754 "round to even". (6.3.1.4)
- The rounding direction when a floating-point number is converted to a narrower floating-point number is IEEE-754 "round to even". (6.3.1.5)
- For floating-point constants that are not exactly representable, the implementation uses the nearest representable value. (6.4.4.2)
- The compiler does not contract float expressions. (6.5)
- The default state for the FENV_ACCESS pragma is off. (7.6.1)
- The TI compiler does not define any additional float exceptions. (7.6, 7.12)
- The default state for the FP_CONTRACT pragma is off. (7.12.2)
- The "inexact" floating-point exception cannot be raised if the rounded result equals the mathematical result. (F.9)

- The "underflow" and "inexact" floating-point exceptions cannot be raised if the result is tiny but not inexact. (F.9)

J.3.7 Arrays and pointers

- When converting a pointer to an integer or vice versa, the pointer is considered an unsigned integer of the same size, and the normal integer conversion rules apply.
- When converting a pointer to an integer or vice versa, if the bitwise representation of the destination can hold all of the bits in the bitwise representation of the source, the bits are copied exactly. (6.3.2.3)
- The size of the result of subtracting two pointers to elements of the same array is the size of ptrdiff_t, which is defined in Section 5.5. (6.5.6)

J.3.8 Hints

- When the optimizer is used, the register storage-class specifier is ignored. When the optimizer is not used, the compiler will preferentially place register storage class objects into registers to the extent possible. The compiler reserves the right to place any register storage class object somewhere other than a register. (6.7.1)
- The inline function specifier is ignored unless the optimizer is used. For other restrictions on inlining, see Section 2.11.2. (6.7.4)

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

- A "plain" int bit-field is treated as a signed int bit-field. (6.7.2, 6.7.2.1)
- In addition to _Bool, signed int, and unsigned int, the compiler allows char, signed char, unsigned char, signed short, unsigned shot, signed long, unsigned long, signed long long, unsigned long long, and enum types as bit-field types. (6.7.2.1)
- Bit-fields may not straddle a storage-unit boundary. (6.7.2.1)
- Bit-fields are allocated in endianness order within a unit. See Section 6.2.2. (6.7.2.1)
- Non-bit-field members of structures are aligned as specified in Section 6.2.1. (6.7.2.1)
- The integer type underlying each enumerated type is described in Section 5.5.1. (6.7.2.2)

J.3.10 Qualifiers

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

J.3.11 Preprocessing directives

- Include directives may have one of two forms, " " or < >. For both forms, the compiler will look for a real file on-disk by that name using the include file search path. See Section 2.5.2. (6.4.7)
- The value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (both are ASCII). (6.10.1)
- The compiler uses the file search path to search for an included < > delimited header file. See Section 2.5.2. (6.10.2)
- The compiler uses the file search path to search for an included " " delimited header file. See Section 2.5.2. (6.10.2)
- There is no arbitrary nesting limit for #include processing. (6.10.2)
- See Section 5.11 for a description of the recognized non-standard pragmas. (6.10.6)
- The date and time of translation are always available from the host. (6.10.8)

J.3.12 Library functions

- Almost all of the library functions required for a hosted implementation are provided by the TI library, with exceptions noted in Section 5.16.1. (5.1.2.1)
- The format of the diagnostic printed by the assert macro is "Assertion failed, (*assertion macro argument*), file *file*, line *line*". (7.2.1.1)
- No strings other than "C" and "" may be passed as the second argument to the setlocale function. (7.11.1.1)
- No signal handling is supported. (7.14.1.1)
- The +INF, -INF, +inf, -inf, NAN, and nan styles can be used to print an infinity or NaN. (7.19.6.1, 7.24.2.1)
- The output for %p conversion in the fprintf or fwprintf function is the same as %x of the appropriate size. (7.19.6.1, 7.24.2.1)
- The termination status returned to the host environment by the abort, exit, or _Exit function is not returned to the host environment. (7.20.4.1, 7.20.4.3, 7.20.4.4)
- The system function is not supported. (7.20.4.6)

J.3.13 Architecture

- The values or expressions assigned to the macros specified in the headers float.h, limits.h, and stdint.h are described along with the sizes and format of integer types are described in Section 5.5. (5.2.4.2, 7.18.2, 7.18.3)
- The number, order, and encoding of bytes in any object are described in Section 6.2.1. (6.2.6.1)
- The value of the result of the sizeof operator is the storage size for each type, in terms of bytes. See Section 6.2.1. (6.5.3.4)

5.2 Characteristics of ARM C++

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

- Complete C++ standard library support, with exceptions noted below.
- Templates
- Exceptions, which are enabled with the --exceptions option; see Section 5.8.
- Run-time type information (RTTI), which can be enabled with the --rtti compiler option.

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

- The compiler does not support embedded C++ run-time-support libraries.
- The library supports wide chars (wchar_t), in that template functions and classes that are defined for char are also available for wchar_t. For example, wide char stream classes wios, wiostream, wstreambuf and so on (corresponding to char classes ios, iostream, streambuf) are implemented. However, there is no low-level file I/O for wide chars. Also, the C library interface to wide char support (through the C++ headers <cwchar> and <cwctype>) is limited as described above in the C library.
- Constant expressions for target-specific types are only partially supported.
- New character types (introduced in the C++11 standard) are not supported.
- Unicode string literals (introduced in the C++11 standard) are not supported.
- Universal character names in literals (introduced in the C++11 standard) are not supported.
- Strong compare and exchange (introduced in the C++11 standard) are not supported.
- Bidirectional fences (introduced in the C++11 standard) are not supported.
- Memory model (introduced in the C++11 standard) is not supported.
- Propagating exceptions (introduced in the C++11 standard) is not supported.
- Thread-local storage (introduced in the C++11 standard) is not supported.
- Dynamic initialization and destruction with concurrency (introduced in the C++11 standard) is not supported.

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

5.3 Using MISRA C 2004

MISRA C is a set of software development guidelines for the C programming language. It promotes best practices in developing safety-related electronic systems in road vehicles and other embedded systems. MISRA C was originally launched in 1998 by the Motor Industry Software Reliability Association, and has since been adopted across a wide variety of industries. A subsequent update to the guidelines was publishes as MISRA C:2004

You can alter your code to work with the MISRA C:2004 rules. The following options and pragmas can be used to enable/disable rules:

- The --check_misra option enables checking of the specified MISRA C:2004 rules. This compiler option must be used if you want to enable further control over checking using the CHECK_MISRA and RESET_MISRA pragmas.
- The CHECK_MISRA pragma enables/disables MISRA C:2004 rules at the source level. See Section 5.11.2.
- The RESET_MISRA pragma resets the specified MISRA C:2004 rules to their state before any CHECK_MISRA pragmas were processed. See Section 5.11.25.

The syntax of the option and the 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 rule numbers to enable or disable.

Example: --check_misra=1.1,1.4,1.5,2.1,2.7,7.1,7.2,8.4

- Enables checking of rules 1.1, 1.4, 1.5, 2.1, 2.7, 7.1, 7.2, and 8.4.

Example: #pragma CHECK_MISRA("-7.1,-7.2,-8.4")

- Disables checking of rules 7.1, 7.2, and 8.4.

A typical use case is to use the --check_misra option on the command line to specify the rules that should be checked in most of your code. Then, use the CHECK_MISRA pragma with a *rulespec* to activate or deactivate certain rules for a particular region of code.

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 Using the ULP Advisor

You can get feedback about your code from the ULP (Ultra-Low Power) Advisor. For a list and descriptions of the ULP rules, see www.ti.com/ulpadvisor. You can enable/disable the rules using any of the following. Using multiple --advice options on the command line is permitted.

- The --advice:power option lets you specify which rules to check.
- The --advice:power_severity option lets you specify whether ULP Advisor rule violations are errors, warnings, remarks, or not reported.
- The CHECK_ULP pragma enables/disables ULP Advisor rules at the source level. This pragma has the same effect as using the --advice:power option. See Section 5.11.3.
- The RESET_ULP pragma resets the specified ULP Advisor rules to their state before any CHECK_ULP pragmas were processed. See Section 5.11.26.

The --advice:power option enables checking specified ULP Advisor rules. The syntax is:

--advice:power={all|none|*rulespec*}

The *rulespec* parameter is a comma-separated list of rule numbers to enable. For example, `--advice:power=1.1,7.2,7.3,7.4` enables rules 1.1, 7.2, 7.3, and 7.4.

The --advice:power_severity option sets the diagnostic severity for ULP Advisor rules. The syntax is:

--advice:power_severity={error|warning|remark|suppress}

The syntax of the pragmas is:

#pragma CHECK_ULP ("{all|none|*rulespec*}**")**

#pragma RESET_ULP ("{all|*rulespec*}**")**

5.5 Data Types

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

The storage and alignment of data types is described in Section 6.2.1.

Table 5-1. ARM C/C++ Data Types

			Range	
Type	**Size**	**Representation**	**Minimum**	**Maximum**
signed char	8 bits	ASCII	-128	127
char (1)	8 bits	ASCII	0 (1)	255 (1)
unsigned char	8 bits	ASCII	0	255
bool, _Bool	8 bits	ASCII	0 (false)	1(true)
short, signed short	16 bits	Binary	-32 768	32 767
unsigned short, wchar_t (2)	16 bits	Binary	0	65 535
int, signed int	32 bits	Binary	-2 147 483 648	2 147 483 647
unsigned int	32 bits	Binary	0	4 294 967 295
long, signed long	32 bits	Binary	-2 147 483 648	2 147 483 647
unsigned long	32 bits	Binary	0	4 294 967 295
long long, signed long long	64 bits(3)	Binary	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits(3)	Binary	0	18 446 744 073 709 551 615
enum (TI_ARM9_ABI and TIABI only) (4)	varies	Binary	varies	varies
float	32 bits	IEEE 32-bit	1.175 494e-38(5)	3.40 282 346e+38
double	64 bits(3)	IEEE 64-bit	2.22 507 385e-308(5)	1.79 769 313e+308
long double	64 bits(3)	IEEE 64-bit	2.22 507 385e-308(5)	1.79 769 313e+308
pointers, references, pointer to data members	32 bits	Binary	0	0xFFFFFFFF

(1) "Plain" char has the same representation as either signed char or unsigned char. The --plain_char option specifies whether "plain" char is signed or unsigned. The default is unsigned.
(2) This is the default type for wchar_t. You can use the --wchar_t option to change the wchar_t type to a 32-bit unsigned int type.
(3) 64-bit data is aligned on a 64-bit boundary.
(4) For details about the size of an enum type, see Section 5.5.1. Also see Table 5-2 for sizes.
(5) Figures are minimum precision.

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

The type of the storage container for an enumerated type is the smallest integer type that contains all the enumerated values. The container types for enumerators are shown in Table 5-2.

Table 5-2. Enumerator Types

Lower Bound Range	Upper Bound Range	Enumerator Type
0 to 255	0 to 255	unsigned char
-128 to 1	-128 to 127	signed char
0 to 65 535	256 to 65 535	unsigned short
-128 to 1	128 to 32 767	short, signed short
-32 768 to -129	-32 768 to 32 767	
0 to 4 294 967 295	2 147 483 648 to 4 294 967 295	unsigned int
-32 768 to -1	32 767 to 2 147 483 647	int, signed int
-2 147 483 648 to -32 769	-2 147 483 648 to 2 147 483 647	
0 to 2 147 483 647	65 536 to 2 147 483 647	

The compiler determines the type based on the range of the lowest and highest elements of the enumerator. For example, the following code results in an enumerator type of int:

```
   enum COLORS
{  green  = -200,
   blue   = 1,
   yellow = 2,
   red    = 60000
};
```

The following code results in an enumerator type of short:

```
   enum COLORS
{  green  = -200,
   blue   = 1,
   yellow = 2,
   red    = 3
};
```

5.5.1 Size of Enum Types

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

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

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

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

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

You may control the strategy for picking enumerated types by using either the --enum_type command line option, or by using an attribute, or both. If you use the --enum_type=packed option (the default), the compiler uses the smallest type it can for the enumerated type. If you use the --enum_type=int option, the underlying type will be int. An enumeration constant with a value outside the int range generates an error.

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

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

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

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

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

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

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

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

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

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

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

5.6 File Encodings and Character Sets

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

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

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

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

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

5.7 Keywords

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

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

- trap
- reentrant
- cregister

5.7.1 The const Keyword

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

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

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

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

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

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

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

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

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

Note that the interrupt function attribute described in Section 5.11.16 is the recommended syntax for declaring interrupt functions.

Functions that handle interrupts follow special register-saving rules and a special return sequence. The implementation stresses safety. The interrupt routine does not assume that the C run-time conventions for the various CPU register and status bits are in effect; instead, it re-establishes any values assumed by the run-time environment. When C/C++ code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any function called by the routine. When you use the __interrupt keyword with the definition of the function, the compiler generates register saves based on the rules for interrupt functions and the special return sequence for interrupts.

You can only use the __interrupt keyword with a function that is defined to return void and that has no parameters. The body of the interrupt function can have local variables and is free to use the stack or global variables. For example:

```
__interrupt void int_handler()
{
    unsigned int flags;
    ...
}
```

The name c_int00 is the C/C++ entry point. This name is reserved for the system reset interrupt. This special interrupt routine initializes the system and calls the main() function. Because it has no caller, c_int00 does not save any registers.

Note

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

5.7.3 The volatile Keyword

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

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

The compiler eliminates redundant memory accesses whenever possible, using data flow analysis to figure out when it is legal. However, some memory accesses may be special in some way that the compiler cannot see, and in such cases you should use the volatile keyword to prevent the compiler from optimizing away something important. The compiler does not optimize out any accesses to variables declared volatile. The number of volatile reads and writes will be exactly as they appear in the C/C++ code, no more and no less and in the same order.

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

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

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

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

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

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

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

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

5.8 C++ Exception Handling

The compiler supports the C++ exception handling features defined by the ANSI/ISO 14882 C++ Standard. See *The C++ Programming Language, Third Edition* by Bjarne Stroustrup. The compiler's --exceptions option enables exception handling. The compiler's default is no exception handling support.

For exceptions to work correctly, all C++ files in the application must be compiled with the --exceptions option, regardless of whether exceptions occur in that file. Mixing exception-enabled and exception-disabled object files and libraries can lead to undefined behavior.

Exception handling requires support in the run-time-support library, which come in exception-enabled and exception-disabled forms; you must link with the correct form. When using automatic library selection (the default), the linker automatically selects the correct library Section 4.3.1.1. If you select the library manually, you must use run-time-support libraries whose name contains _eh if you enable exceptions.

Using the --exceptions option causes the compiler to insert exception handling code. This code will increase the size of the program , but EABI does not increase the code size much, and has a minimal execution time cost if exceptions are never thrown. It slightly increases the data size for the exception-handling tables.

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

5.9 Register Variables and Parameters

The C/C++ compiler allows the use of the keyword register on global and local register variables and parameters. This section describes the compiler implementation for this qualifier.

5.9.1 Local 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.

5.9.2 Global Register Variables

The C/C++ compiler extends the C language by adding a special convention to the register storage class specifier to allow the allocation of global registers. This special global declaration has the form:

register *type regid*

The *regid* parameter can be __R5, __R6, or __R9. The identifiers _ _R5, _ _R6, and _ _R9 are each bound to their corresponding register R5, R6 and R9, respectively.

When you use this declaration at the file level, the register is permanently reserved from any other use by the optimizer and code generator for that file. You cannot assign an initial value to the register. You can use a #define directive to assign a meaningful name to the register; for example:

```
register struct data_struct *__R5
#define data_pointer __R5
data_pointer->element;
data_pointer++;
```

There are two reasons that you would be likely to use a global register variable:

- You are using a global variable throughout your program, and it would significantly reduce code size and execution speed to assign this variable to a register permanently.
- You are using an interrupt service routine that is called so frequently that it would significantly reduce execution speed if the routine did not have to save and restore the register(s) it uses every time it is called.

You need to consider very carefully the implications of reserving a global register variable. Registers are a precious resource to the compiler, and using this feature indiscriminately may result in poorer code.

You also need to consider carefully how code with a globally declared register variable interacts with other code, including library functions, that does not recognize the restriction placed on the register.

Because the registers that can be global register variables are save-on-entry registers, a normal function call and return does not affect the value in the register and neither does a normal interrupt. However, when you mix code that has a globally declared register variable with code that does not have the register reserved, it is still possible for the value in the register to become corrupted. To avoid the possibility of corruption, you must follow these rules:

- Functions that alter global register variables cannot be called by functions that are not aware of the global register. Use the -r shell option to reserve the register in code that is not aware of the global register declaration. You must be careful if you pass a pointer to a function as an argument. If the passed function alters the global register variable and the called function saves the register, the value in the register will be corrupted.
- You cannot access a global register variable in an interrupt service routine unless you recompile all code, including all libraries, to reserve the register. This is because the interrupt routine can be called from any point in the program.
- The longjmp ( ) function restores global register variables to the values they had at the setjmp ( ) location. If this presents a problem in your code, you must alter the code for the function and recompile rts.src.

  The -r *register* compiler command-line option allows you to prevent the compiler from using the named register. This lets you reserve the named register in modules that do not have the global register variable declaration, such as the run-time-support libraries, if you need to compile the modules to prevent some of the above occurrences.

5.10 The __asm Statement

The C/C++ compiler can embed assembly language instructions or directives directly into the assembly language output of the compiler. This capability is an extension to the C/C++ language implemented through the ***__asm*** keyword. The __asm keyword provides access to hardware features that C/C++ cannot provide.

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

Using __asm is syntactically performed as a call to a function named __asm, with one string constant argument:

__asm(" *assembler text* **");**

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

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

The *naked* function attribute can be used to identify functions that are written as embedded assembly functions using `__asm` statements. See Section 5.17.2.

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

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

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

For more information, refer to Section 6.6.5.

Note

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

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

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

5.11 Pragma Directives

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

- CALLS (See Section 5.11.1)
- CHECK_MISRA (See Section 5.11.2)
- CHECK_ULP (See Section 5.11.3)
- CODE_SECTION (See Section 5.11.4)
- CODE_STATE (See Section 5.11.5)
- DATA_ALIGN (See Section 5.11.6)
- DATA_SECTION (See Section 5.11.7)
- diag_suppress, diag_remark, diag_warning, diag_error, diag_default, diag_push, diag_pop (See Section 5.11.8)
- DUAL_STATE (See Section 5.11.9)
- FORCEINLINE (See Section 5.11.10)
- FORCEINLINE_RECURSIVE (See Section 5.11.11)
- FUNC_ALWAYS_INLINE (See Section 5.11.12)
- FUNC_CANNOT_INLINE (See Section 5.11.13)
- FUNC_EXT_CALLED (See Section 5.11.14)
- FUNCTION_OPTIONS (See Section 5.11.15)
- INTERRUPT (See Section 5.11.16)
- LOCATION (See Section 5.11.17)
- MUST_ITERATE (See Section 5.11.18)
- NOINIT (See Section 5.11.19)
- NOINLINE (See Section 5.11.20)
- NO_HOOKS (See Section 5.11.21)
- once (See Section 5.11.22)
- pack (See Section 5.11.23)
- PERSISTENT (See Section 5.11.19)
- PROB_ITERATE (See Section 5.11.24)
- RESET_MISRA (See Section 5.11.25)
- RESET_ULP (See Section 5.11.26)
- RETAIN (See Section 5.11.27)
- SET_CODE_SECTION (See Section 5.11.28)
- SET_DATA_SECTION (See Section 5.11.28)
- SWI_ALIAS (See Section 5.11.29)
- TASK (See Section 5.11.30)
- UNROLL (See Section 5.11.31)
- WEAK (See Section 5.11.32)

The arguments *func* and *symbol* cannot be defined or declared inside the body of a function. You must specify the pragma outside the body of a function, and pragma specifications must occur before any declaration, definition, or reference to the func or symbol argument. If you do not follow these rules, the compiler issues a warning and may ignore the pragma.

For pragmas that apply to functions or symbols, the syntax differs between C and C++.

- In C, you must supply the name of the object or function to which you are applying the pragma as the first argument. Because the entity operated on is specified, a pragma in C can appear some distance way from the definition of that entity.
- In C++, pragmas are positional. They do not name the entity on which they operate as an argument. Instead, they always operate on the next entity defined after the pragma.

5.11.1 The CALLS Pragma

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

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

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

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

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

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

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

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

The GCC-style "calls" function attribute (see Section 5.17.2), which has the same effect as the CALLS pragma, has the following syntax:

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

5.11.2 The CHECK_MISRA Pragma

The CHECK_MISRA pragma enables/disables MISRA C:2004 rules at the source level. The compiler option --check_misra must be used to enable checking in order for this pragma to function at the source level.

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 rule numbers. See Section 5.3 for details.

The RESET_MISRA pragma can be used to reset any CHECK_MISRA pragmas; see Section 5.11.25.

5.11.3 The CHECK_ULP Pragma

The CHECK_ULP pragma enables/disables ULP Advisor rules at the source level. This pragma has the same effect as using the --advice:power option.

The syntax of the pragma in C is:

#pragma CHECK_ULP (" {all|none|*rulespec*} **")**

The *rulespec* parameter is a comma-separated list of rule numbers. See Section 5.4 for the syntax. See www.ti.com/ulpadvisor for a list of rules.

The RESET_ULP pragma can be used to reset any CHECK_ULP pragmas; see Section 5.11.26.

5.11.4 The CODE_SECTION Pragma

The CODE_SECTION pragma allocates space for the *symbol* in C, or the next symbol declared in C++, in a section named *section name*. The CODE_SECTION pragma is useful if you have code objects that you want to link into an area separate from the .text section. The CODE_SECTION pragma has the same effect as using the GCC-style `section` function attribute. See Section 5.17.2.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

The following example demonstrates the use of the CODE_SECTION pragma.

Using the CODE_SECTION Pragma in C

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

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

```
            .sect     "my_sect"
            .align    4
            .state32
            .global   fn
;******************************************************************************
;* FUNCTION NAME: fn                                                          *
;*                                                                            *
;*   Regs Modified     : SP                                                   *
;*   Regs Used         : A1,SP                                                *
;*   Local Frame Size  : 0 Args + 4 Auto + 0 Save = 4 byte                    *
;******************************************************************************
fn:
;* ---------------------------------------------------------------------------*
        SUB       SP, SP, #8
        STR       A1, [SP, #0]          ; |4|
        ADD       SP, SP, #8
        BX        LR
```

5.11.5 The CODE_STATE Pragma

The CODE_STATE pragma overrides the compilation state of a file, at the function level. For example, if a file is compiled in thumb mode, but you want a function in that file to be compiled in 32-bit mode, you would add this pragma in the file. The compilation state for the function is changed to 16-bit mode (thumb) or 32-bit mode.

The syntax of the pragma is C is:

#pragma CODE_STATE (*function* **,** {16|32} **)**

The syntax of the pragma in C++ is:

#pragma CODE_STATE (*code state* **)**

5.11.6 The DATA_ALIGN Pragma

The DATA_ALIGN pragma aligns the *symbol* in C, or the next symbol declared in C++, to an alignment boundary. The alignment boundary is the maximum of the symbol's default alignment value or the value of the *constant* in bytes. The constant must be a power of 2. The maximum alignment is 32768.

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

Using the DATA_ALIGN pragma has the same effect as using the GCC-style `aligned` variable attribute. See Section 5.17.4.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

5.11.7 The DATA_SECTION Pragma

The DATA_SECTION pragma allocates space for the *symbol* in C, or the next symbol declared in C++, in a section named *section name*. This pragma is useful if you have data objects that you want to link into an area separate from the .bss section.

Using the DATA_SECTION pragma has the same effect as using the GCC-style `section` variable attribute. See Section 5.17.4.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

Example 5-1 through Example 5-3 demonstrate the use of the DATA_SECTION pragma.

Example 5-1. Using the DATA_SECTION Pragma C Source File

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

Example 5-2. Using the DATA_SECTION Pragma C++ Source File

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

Example 5-3. Using the DATA_SECTION Pragma Assembly Source File

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

5.11.8 The Diagnostic Message Pragmas

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

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

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

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

Notice that the names of these pragmas are in lowercase.

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

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

5.11.9 The DUAL_STATE Pragma

By default (that is, without the compiler -md option), all functions with external linkage support dual-state interworking. This support assumes that most calls do not require a state change and are therefore optimized (in terms of code size and execution speed) for calls not requiring a state change. Using the DUAL_STATE pragma does not change the functionality of the dual-state support, but it does assert that calls to the applied function often require a state change. Therefore, such support is optimized for state changes.

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. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

#pragma DUAL_STATE (*func* **)**

The syntax of the pragma in C++ is:

#pragma DUAL_STATE

For more information on dual-state interworking, see Section 6.11.

5.11.10 The FORCEINLINE Pragma

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

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

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

```
#pragma FORCEINLINE
```

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

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

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

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

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

5.11.11 The FORCEINLINE_RECURSIVE Pragma

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

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

```
#pragma FORCEINLINE_RECURSIVE
```

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

5.11.12 The FUNC_ALWAYS_INLINE Pragma

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

The compiler only inlines a function if it is legal to inline the function. Functions are never inlined if the compiler is invoked with the --opt_level=off option. A function can be inlined even if the function is not declared with the inline keyword. A function can be inlined even if the compiler is not invoked with any --opt_level command-line option. See Section 2.11 for details about interaction between various types of inlining.

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

The FUNC_ALWAYS_INLINE pragma has the same effect as using the GCC-style `always_inline` function attribute. See Section 5.17.2.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma FUNC_ALWAYS_INLINE

The following example uses this pragma:

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

Note

Use Caution with the FUNC_ALWAYS_INLINE Pragma

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

5.11.13 The FUNC_CANNOT_INLINE Pragma

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

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

The FUNC_CANNOT_INLINE pragma has the same effect as using the GCC-style `noinline` function attribute. See Section 5.17.2.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma FUNC_CANNOT_INLINE

5.11.14 The FUNC_EXT_CALLED Pragma

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

The FUNC_EXT_CALLED pragma specifies that the optimizer should keep these C functions or any functions these C/C++ functions call. These functions act as entry points into C/C++. The pragma must appear before any declaration or reference to the function to keep. In C, the argument *func* is the name of the function to keep. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

```
#pragma FUNC_EXT_CALLED
```

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

When you use program-level optimization, you may need to use the FUNC_EXT_CALLED pragma with certain options. See Section 3.4.2.

5.11.15 The FUNCTION_OPTIONS Pragma

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

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

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

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

5.11.16 The INTERRUPT Pragma

The INTERRUPT pragma enables you to handle interrupts directly with C code. The pragma specifies that the function is an interrupt. The type of interrupt is specified by the pragma; the IRQ (interrupt request) interrupt type is assumed if none is given.

The syntax of the pragma in C is:

#pragma INTERRUPT (*func* [**,***interrupt_type*] **)**

The syntax of the pragma in C++ is:

#pragma INTERRUPT [**(***interrupt_type***)**]
void *func* **(** void **)**

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

__attribute__((interrupt [**(***"interrupt_type"***)**] **))** void *func* **(** void **)**

In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared. The optional argument *interrupt_type* specifies an interrupt type. The registers that are saved and the return sequence depend upon the interrupt type. If the interrupt type is omitted from the interrupt pragma, the interrupt type IRQ is assumed. These are the valid interrupt types:

Interrupt Type	Description
DABT	Data abort
FIQ	Fast interrupt request
IRQ	Interrupt request
PABT	Prefetch abort
RESET	System reset
SWI	Software interrupt
UDEF	Undefined instruction

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.

For the Cortex-M architectures, the *interrupt_type* can be nothing (default) or SWI. The hardware performs the necessary saving and restoring of context for interrupts. Therefore, the compiler does not distinguish between the different interrupt types. The only exception is for software interrupts (SWIs) which are allowed to have arguments (for Cortex-M architectures, C SWI handlers cannot return values).

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

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

Note

Hwi Objects and the INTERRUPT Pragma: The INTERRUPT pragma must not be used when SYS/BIOS Hwi objects are used in conjunction with C functions. The Hwi_enter/Hwi_exit macros and the Hwi dispatcher contain this functionality, and the use of the C modifier can cause negative results.

5.11.17 The LOCATION Pragma

The compiler supports the ability to specify the run-time address of a variable at the source level. This can be accomplished with the LOCATION pragma or the GCC-style location attribute. The LOCATION pragma has the same effect as using the GCC-style `location` function attribute. See Section 5.17.2.

The syntax of the pragma in C is:

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

The syntax of the pragmas in C++ is:

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

The syntax of the GCC-style attribute (see Section 5.17.4) is:

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

The NOINIT pragma may be used in conjunction with the LOCATION pragma to map variables to special memory locations; see Section 5.11.19.

5.11.18 The MUST_ITERATE Pragma

The MUST_ITERATE pragma specifies to the compiler certain properties of a loop. When you use this pragma, you are guaranteeing to the compiler that a loop executes a specific number of times or a number of times within a specified range.

Any time the UNROLL pragma is applied to a loop, MUST_ITERATE should be applied to the same loop. For loops the MUST_ITERATE pragma's third argument, multiple, is the most important and should always be specified.

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

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

5.11.18.1 The MUST_ITERATE Pragma Syntax

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

```
#pragma MUST_ITERATE ( min, max, multiple )
```

The C++ syntax for the corresponding attribute is as follows. No C attribute syntax is available.

```
[[TI::must_iterate( min, max, multiple )]]
```

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

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

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

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

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

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

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

The following example uses the must_iterate C++ attribute syntax:

```
void myFunc (int *a, int *b, int * restrict c, int n)
{
    ...
    [[TI::must_iterate(32, 1024, 16)]]
    for (int i = 0; i < n; i++)
    {
        c[i] = a[i] + b[i];
    }
    ...
}
```

5.11.18.2 Using MUST_ITERATE to Expand Compiler Knowledge of Loops

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

```
#pragma MUST_ITERATE(10,10)
for(i = 0; i < trip_count; i++)  { ...
```

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

MUST_ITERATE can specify a range for the trip count as well as a factor of the trip count. The following example tells the compiler that the loop executes between 8 and 48 times and the trip_count variable is a multiple of 8 (8, 16, 24, 32, 40, 48). The multiple argument allows the compiler to unroll the loop.

```
#pragma MUST_ITERATE(8, 48, 8)
for(i = 0; i < trip_count; i++)  { ...
```

You should consider using MUST_ITERATE for loops with complicated bounds. In the following example, the compiler would have to generate a divide function call to determine, at run time, the number of iterations performed.

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

The compiler will not do the above. In this case, using MUST_ITERATE to specify that the loop always executes eight times allows the compiler to attempt to generate a loop:

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

5.11.19 The NOINIT and PERSISTENT Pragmas

Global and static variables are zero-initialized by default. However, in applications that use non-volatile memory, it may be desirable to have variables that are not initialized. Noinit variables are global or static variables that are not zero-initialized at startup or reset.

Variables can be declared as noinit or persistent using either pragmas or variable attributes. See Section 5.17.4 for information about using variable attributes in declarations.

Noinit and persistent variables behave identically with the exception of whether or not they are initialized at load time.

- The NOINIT pragma may be used only with uninitialized variables. It prevents such variables from being set to 0 during a reset. It may be used in conjunction with the LOCATION pragma to map variables to special memory locations, like memory-mapped registers, without generating unwanted writes.
- The PERSISTENT pragma may be used only with statically-initialized variables. It prevents such variables from being initialized during a reset. Persistent variables disable startup initialization; they are given an initial value when the code is loaded, but are never again initialized.

By default, noinit or persistent variables are placed in sections named `.TI.noinit` and `.TI.persistent` , respectively. The location of these sections is controlled by the linker command file. Typically .TI.persistent sections are placed in FRAM for devices that support FRAM and .TI.noinit sections are placed in RAM.

Note

When using these pragmas in non-volatile FRAM memory, the memory region could be protected against unintended writes through the device's Memory Protection Unit. Some devices have memory protection enabled by default. Please see the information about memory protection in the datasheet for your device. If the Memory Protection Unit is enabled, it first needs to be disabled before modifying the variables.

If you are using non-volatile RAM, you can define a persistent variable with an initial value of zero loaded into RAM. The program can increment that variable over time as a counter, and that count will not disappear if the device loses power and restarts, because the memory is non-volatile and the boot routines do not initialize it back to zero. For example:

```
#pragma PERSISTENT(x)
#pragma location = 0xC200   // memory address in RAM
int x = 0;
void main() {
    run_init();
    while (1) {
        run_actions(x);
        __delay_cycles(1000000);
        x++;
    }
}
```

The syntax of the pragmas in C is:

```
#pragma NOINIT ( x )
int x ;
```

```
#pragma PERSISTENT ( x )
int x =10;
```

The syntax of the pragmas in C++ is:

```
#pragma NOINIT
int x ;
```

```
#pragma PERSISTENT
int x =10;
```

The syntax of the GCC attributes is:

```
int x __attribute__((noinit));
```

```
int x __attribute__((persistent)) = 0;
```

5.11.20 The NOINLINE Pragma

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

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

```
#pragma NOINLINE
```

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

5.11.21 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 2.15 for details on entry and exit hooks.

5.11.22 The once Pragma

The once pragma tells the C preprocessor to ignore a #include directive if that header file has already been included. For example, this pragma may be used if header files contain definitions, such as struct definitions, that would cause a compilation error if they were executed more than once.

This pragma should be used at the beginning of a header file that should only be included once. For example:

```
// hdr.h
#pragma once
#warn You will only see this message one time
struct foo
{
    int member;
};
```

This pragma is not part of the C or C++ standard, but it is a widely-supported preprocessor directive. Note that this pragma does not protect against the inclusion of a header file with the same contents that has been copied to another directory.

5.11.23 The pack Pragma

The pack pragma can be used to control the alignment of fields within a class, struct, or union type. The syntax of the pragma in C/C++ can be any of the following.

#pragma pack (*n*)

The above form of the pack pragma affects all class, struct, or union type declarations that follow this pragma in a file. It forces the maximum alignment of each field to be the value specified by *n*. Valid values for *n* are 1, 2, 4, 8, and 16 bytes.

#pragma pack (push, *n* **)**
#pragma pack (pop)

The above form of the pack pragma affects only class, struct, and union type declarations between push and pop directives. (A pop directive with no prior push results in a warning diagnostic from the compiler.) The maximum alignment of all fields declared is *n*. Valid values for *n* are 1, 2, 4, 8, and 16 bytes.

#pragma pack (show)

The above form of the pack pragma sends a warning diagnostic to stderr to record the current state of the pack pragma stack. You can use this form while debugging.

For more about packed fields, see Section 5.17.5.

5.11.24 The PROB_ITERATE Pragma

The PROB_ITERATE pragma specifies to the compiler certain properties of a loop. You assert that these properties are true in the common case. The PROB_ITERATE pragma aids the compiler in choosing the best loops and loop transformations (that is, software pipelining and nested loop transformations). PROB_ITERATE is useful only when the MUST_ITERATE pragma is not used or the PROB_ITERATE parameters are more constraining than the MUST_ITERATE parameters.

No statements are allowed between the PROB_ITERATE pragma and the for, while, or do-while loop to which it applies. However, other pragmas, such as UNROLL and MUST_ITERATE, may appear between the PROB_ITERATE pragma and the loop. The syntax of the pragma for C and C++ is:

#pragma PROB_ITERATE(*min* **,** *max* **)**

The C++ syntax for the corresponding attribute is as follows. No C attribute syntax is available. See Section 5.11.18.1 for an example that uses similar syntax.

[[TI::prob_iterate(*min*, *max* **)]]**

Where min and max are the minimum and maximum trip counts of the loop in the common case. The trip count is the number of times a loop iterates. Both arguments are optional.

For example, PROB_ITERATE could be applied to a loop that executes for eight iterations in the majority of cases (but sometimes may execute more or less than eight iterations):

```
#pragma PROB_ITERATE(8, 8)
```

If only the minimum expected trip count is known (say it is 5), the pragma would look like this:

```
#pragma PROB_ITERATE(5)
```

If only the maximum expected trip count is known (say it is 10), the pragma would look like this:

```
#pragma PROB_ITERATE(, 10)  /* Note the blank field for min */
```

5.11.25 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 5.11.2) 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 --check_misra compiler command-line option must be used to enable MISRA C:2004 rule checking in order for this pragma to function at the source level.

The syntax of the pragma in C is:

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

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

5.11.26 The RESET_ULP Pragma

The RESET_ULP pragma resets the specified ULP Advisor rules to the state they were before any CHECK_ULP pragmas (see Section 5.11.3) were processed. For instance, if a rule was enabled on the command line but disabled in the source, the RESET_ULP pragma resets it to enabled. This pragma accepts the same format as the --advice:power option, except for the "none" keyword.

The syntax of the pragma in C is:

#pragma RESET_ULP (" {all|*rulespec*} **")**

The *rulespec* parameter is a comma-separated list of rule numbers. See Section 5.4 for details. See www.ti.com/ulpadvisor for a list of rules.

5.11.27 The RETAIN Pragma

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

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

The RETAIN pragma has the same effect as using the `retain` function or variable attribute. See Section 5.17.2 and Section 5.17.4, respectively.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma RETAIN

5.11.28 The SET_CODE_SECTION and SET_DATA_SECTION Pragmas

These pragmas can be used to set the section for all declarations below the pragma.

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

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

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

In the Setting Section With SET_DATA_SECTION Pragma example, x and y are put in the section mydata. To reset the current section to the default used by the compiler, a blank parameter should be passed to the pragma. An easy way to think of the pragma is that it is like applying the CODE_SECTION or DATA_SECTION pragma to all symbols below it.

Setting Section With SET_DATA_SECTION Pragma

```
#pragma SET_DATA_SECTION("mydata")
int x;
int y;
#pragma SET_DATA_SECTION()
```

The pragmas apply to both declarations and definitions. If applied to a declaration and not the definition, the pragma that is active at the declaration is used to set the section for that symbol. Here is an example:

Setting a Section With SET_CODE_SECTION Pragma

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

In the Setting a Section With SET_CODE_SECTION Pragma example, func1 is placed in section func1. If conflicting sections are specified at the declaration and definition, a diagnostic is issued.

The current CODE_SECTION and DATA_SECTION pragmas and GCC attributes can be used to override the SET_CODE_SECTION and SET_DATA_SECTION pragmas. For example:

Overriding SET_DATA_SECTION Setting

```
#pragma DATA_SECTION(x, "x_data")
#pragma SET_DATA_SECTION("mydata")
int x;
int y;
#pragma SET_DATA_SECTION()
```

In the Overriding SET_DATA_SECTION Setting example, x is placed in x_data and y is placed in mydata. No diagnostic is issued for this case.

The pragmas work for both C and C++. In C++, the pragmas are ignored for templates and for implicitly created objects, such as implicit constructors and virtual function tables.

The SET_DATA_SECTION pragma takes precedence over the --gen_data_subsections=on option if it is used.

5.11.29 The SWI_ALIAS Pragma

The SWI_ALIAS pragma allows you to refer to a particular software interrupt as a function name and to invocations of the software interrupt as function calls. Since the function name is simply an alias for the software interrupt, no function definition exists for the function name.

The syntax of the pragma in C is:

#pragma SWI_ALIAS(*func* **,** *swi_number* **)**

The syntax of the pragma in C++ is:

#pragma SWI_ALIAS(*swi_number* **)**

Calls to the applied function are compiled as software interrupts whose number is *swi_number*. The *swi_number* variable must be an integer constant.

A function prototype must exist for the alias and it must occur after the pragma and before the alias is used. Software interrupts whose number is not known until run time are not supported.

For information about using the GCC function attribute syntax to declare function aliases, see Section 5.17.2.

For more information about using software interrupts, including restrictions on passing arguments and register usage, see Section 6.7.5.

Using the SWI_ALIAS Pragma C Source File

```
#pragma SWI_ALIAS(put, 48)    /* #pragma SWI_ALIAS(48) for C++ */
int  put  (char *key, int value);
void error();
main()
{
   if (!put("one", 1))  /* calling "put" invokes SWI #48 with 2 arguments */
      error();          /* and returns a result.                          */
}
```

Generated Assembly File

```
;***************************************************************
;* FUNCTION DEF: _main                                         *
;***************************************************************
_main:
        STMFD     SP!, {LR}
        ADR       A1, SL1
        MOV       A2, #1
        SWI       #48            ; SWI #48 is generated for the function call
        CMP       A1, #0
        BLEQ      _error
        MOV       A1, #0
        LDMFD     SP!, {PC}
SL1:    .string "one",0
```

5.11.30 The TASK Pragma

The TASK pragma specifies that the function to which it is applied is a task. Tasks are functions that are called but never return. Typically, they consist of an infinite loop that simply dispatches other activities. Because they never return, there is no need to save (and therefore restore) registers that would otherwise be saved and restored. This can save RAM space, as well as some code space.

The syntax of the pragma in C is:

```
#pragma TASK( func )
```

The syntax of the pragma in C++ is:

```
#pragma TASK
```

5.11.31 The UNROLL Pragma

The UNROLL pragma specifies to the compiler how many times a loop should be unrolled. The optimizer must be invoked (use --opt_level=[1|2|3] or -O1, -O2, or -O3) in order for pragma-specified loop unrolling to take place. The compiler has the option of ignoring this pragma.

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

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

```
#pragma UNROLL( n )
```

The C++ syntax for the corresponding attribute is as follows. No C attribute syntax is available. See Section 5.11.18.1 for an example that uses similar syntax.

```
[[TI::unroll( n )]]
```

If possible, the compiler unrolls the loop so there are *n* copies of the original loop. The compiler only unrolls if it can determine that unrolling by a factor of *n* is safe. In order to increase the chances the loop is unrolled, the compiler needs to know certain properties:

- The loop iterates a multiple of *n* times. This information can be specified to the compiler via the multiple argument in the MUST_ITERATE pragma.
- The smallest possible number of iterations of the loop
- The largest possible number of iterations of the loop

The compiler can sometimes obtain this information itself by analyzing the code. However, sometimes the compiler can be overly conservative in its assumptions and therefore generates more code than is necessary when unrolling. This can also lead to not unrolling at all. Furthermore, if the mechanism that determines when the loop should exit is complex, the compiler may not be able to determine these properties of the loop. In these cases, you must tell the compiler the properties of the loop by using the MUST_ITERATE pragma.

Specifying #pragma UNROLL(1) asks that the loop not be unrolled. Automatic loop unrolling also is not performed in this case.

If multiple UNROLL pragmas are specified for the same loop, it is undefined which pragma is used, if any.

5.11.32 The WEAK Pragma

The WEAK pragma gives weak binding to a symbol.

The syntax of the pragma in C is:

#pragma WEAK (*symbol* **)**

The syntax of the pragma in C++ is:

#pragma WEAK

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

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

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

The WEAK pragma has the same effect as using the `weak` function or variable attribute. See Section 5.17.2 and Section 5.17.4, respectively.

5.12 The _Pragma Operator

The ARM 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.

5.13 Application Binary Interface

An Application Binary Interface (ABI) defines how functions that are written separately and compiled or assembled separately can work together. This involves standardizing data type storage, register conventions, and function structure and calling conventions. It should define linkname generation from C symbols. It defines the object file format and the debug format. It should document how the system is initialized. In the case of C++ it defines C++ name mangling and exception handling support.

The COFF ABI is not supported in v15.6.0.STS and later versions of the TI Code Generation Tools. If you want to produce COFF output files, please use v5.2 of the ARM tools and see SPRU151J.

The ARM ABIv2 has become an industry standard for the ARM architecture. It has these advantages:

- It enables interlinking of objects built with different tool chains. For example, this enables a library built with RVCT to be linked in with an application built with the ARM 4.6 toolset.
- It is well documented. The complete ARM ABI specifications are in the ARM Information Center.
- It is modern. EABI requires ELF object file format which enables supporting modern language features like early template instantiation and export inline functions support.

ARM ABIv2 allows a vendor to define the system initialization in the bare-metal mode. TI-specific information on EABI mode is described in Section 6.10.3. The __TI_EABI_ASSEMBLER predefined symbol is set to 1 if compiling for EABI.

5.14 ARM Instruction Intrinsics

Assembly instructions can be generated using the intrinsics in the following tables. Table 5-3 shows which intrinsics are available on the different ARM targets. Table 5-4 shows the calling syntax for each intrinsic, along with the corresponding assembly instruction and a description. Additional intrinsices for getting and setting the CPSR register and to enable/disable interrupts are provided in Section 6.8.1.

Table 5-3. ARM Intrinsic Support by Target

C/C++ Compiler Intrinsic	ARM V5e (ARM9E)	ARM V6 (ARM11)	ARM V6M0 (Cortex-M0)	ARM V7M3 (Cortex-M3)	ARM V7M4 (Cortex-M4)	ARM V7R (Cortex-R4)	ARM V7A8 (Cortex-A8)
__clz	yes	yes		yes	yes	yes	yes
__delay_cycles			yes	yes	yes	yes	
__get_MSP			yes	yes	yes		
__get_PRIMASK			yes	yes	yes		
__ldrex		yes		yes	yes	yes	yes
__ldrexb		yes		yes	yes	yes	yes
__ldrexd		yes				yes	yes
__ldrexh		yes		yes	yes	yes	yes
__MCR	yes	yes		yes	yes	yes	yes
__MRC	yes	yes		yes	yes	yes	yes
__ nop	yes	yes	yes	yes	yes	yes	yes
_norm	yes	yes		yes	yes	yes	yes
__rev		yes	yes		yes	yes	yes
__rev16		yes	yes		yes	yes	yes
__revsh		yes	yes		yes	yes	yes
__rbit		yes			yes	yes	yes
__ror	yes	yes	yes	yes	yes	yes	yes
_pkhbt		yes			yes	yes	yes
_pkhtb		yes			yes	yes	yes
_qadd16		yes			yes	yes	yes
_qadd8		yes			yes	yes	yes
_qaddsubx		yes			yes	yes	yes

Table 5-3. ARM Intrinsic Support by Target (continued)

C/C++ Compiler Intrinsic	ARM V5e (ARM9E)	ARM V6 (ARM11)	ARM V6M0 (Cortex-M0)	ARM V7M3 (Cortex-M3)	ARM V7M4 (Cortex-M4)	ARM V7R (Cortex-R4)	ARM V7A8 (Cortex-A8)
_qsub16		yes			yes	yes	yes
_qsub8		yes			yes	yes	yes
_qsubaddx		yes			yes	yes	yes
_sadd	yes	yes			yes	yes	yes
_sadd16		yes			yes	yes	yes
_sadd8		yes			yes	yes	yes
_saddsubx		yes			yes	yes	yes
_sdadd	yes	yes			yes	yes	yes
_sdsub	yes	yes			yes	yes	yes
_sel		yes			yes	yes	yes
__set_MSP			yes	yes	yes		
__set_PRIMASK			yes	yes	yes		
_shadd16		yes			yes	yes	yes
_shadd8		yes			yes	yes	yes
_shsub16		yes			yes	yes	yes
_shsub8		yes			yes	yes	yes
_smac	yes	yes			yes	yes	yes
_smlabb	yes	yes			yes	yes	yes
_smlabt	yes	yes			yes	yes	yes
_smlad		yes			yes	yes	yes
_smladx		yes			yes	yes	yes
_smlalbb	yes	yes			yes	yes	yes
_smlalbt	yes	yes			yes	yes	yes
_smlald		yes			yes	yes	yes
_smlaldx		yes			yes	yes	yes
_smlaltb	yes	yes			yes	yes	yes
_smlaltt	yes	yes			yes	yes	yes
_smlatb	yes	yes			yes	yes	yes
_smlatt	yes	yes			yes	yes	yes
_smlawb	yes	yes			yes	yes	yes
_smlawt	yes	yes			yes	yes	yes
_smlsd		yes			yes	yes	yes
_smlsdx		yes			yes	yes	yes
_smlsld		yes			yes	yes	yes
_smlsldx		yes			yes	yes	yes
_smmla		yes			yes	yes	yes
_smmlar		yes			yes	yes	yes
_smmls		yes			yes	yes	yes
_smmlsr		yes			yes	yes	yes
_smmul		yes			yes	yes	yes
_smmulr		yes			yes	yes	yes
_smuad		yes			yes	yes	yes
_smuadx		yes			yes	yes	yes
_smusd		yes			yes	yes	yes

Table 5-3. ARM Intrinsic Support by Target (continued)

C/C++ Compiler Intrinsic	ARM V5e (ARM9E)	ARM V6 (ARM11)	ARM V6M0 (Cortex-M0)	ARM V7M3 (Cortex-M3)	ARM V7M4 (Cortex-M4)	ARM V7R (Cortex-R4)	ARM V7A8 (Cortex-A8)
_smusdx		yes			yes	yes	yes
_smpy	yes	yes			yes	yes	yes
_smsub	yes	yes			yes	yes	yes
_smulbb	yes	yes			yes	yes	yes
_smulbt	yes	yes			yes	yes	yes
_smultb	yes	yes			yes	yes	yes
_smultt	yes	yes			yes	yes	yes
_smulwb	yes	yes			yes	yes	yes
_smulwt	yes	yes			yes	yes	yes
__sqrt	yes	yes				yes	yes
__sqrtf	yes	yes			yes	yes	yes
_ssat16		yes			yes	yes	yes
_ssata	yes	yes		yes	yes	yes	yes
_ssatl	yes	yes		yes	yes	yes	yes
_ssub	yes	yes			yes	yes	yes
_ssub16		yes			yes	yes	yes
_ssub8		yes			yes	yes	yes
_ssubaddx		yes			yes	yes	yes
__strex		yes		yes	yes	yes	yes
__strexb		yes		yes	yes	yes	yes
__strexd		yes				yes	yes
__strexh		yes		yes	yes	yes	yes
_subc	yes	yes			yes	yes	yes
_sxtab		yes			yes	yes	yes
_sxtab16		yes			yes	yes	yes
_sxtah		yes			yes	yes	yes
_sxtb	yes	yes		yes	yes	yes	yes
_sxtb16		yes			yes	yes	yes
_sxth	yes	yes		yes	yes	yes	yes
_uadd16		yes			yes	yes	yes
_uadd8		yes			yes	yes	yes
_uaddsubx		yes			yes	yes	yes
_uhadd16		yes			yes	yes	yes
_uhadd8		yes			yes	yes	yes
_uhsub16		yes			yes	yes	yes
_uhsub8		yes			yes	yes	yes
_umaal		yes			yes	yes	yes
_uqadd16		yes			yes	yes	yes
_uqadd8		yes			yes	yes	yes
_uqaddsubx		yes			yes	yes	yes
_uqsub16		yes			yes	yes	yes
_uqsub8		yes			yes	yes	yes
_uqsubaddx		yes			yes	yes	yes
_usad8		yes			yes	yes	yes

Table 5-3. ARM Intrinsic Support by Target (continued)

C/C++ Compiler Intrinsic	ARM V5e (ARM9E)	ARM V6 (ARM11)	ARM V6M0 (Cortex-M0)	ARM V7M3 (Cortex-M3)	ARM V7M4 (Cortex-M4)	ARM V7R (Cortex-R4)	ARM V7A8 (Cortex-A8)
_usat16		yes			yes	yes	yes
_usata	yes	yes		yes	yes	yes	yes
_usatl	yes	yes		yes	yes	yes	yes
_usub16		yes			yes	yes	yes
_usub8		yes			yes	yes	yes
_usubaddx		yes			yes	yes	yes
_uxtab		yes			yes	yes	yes
_uxtab16		yes			yes	yes	yes
_uxtah		yes			yes	yes	yes
_uxtb	yes	yes		yes	yes	yes	yes
_uxtb16		yes			yes	yes	yes
_uxth	yes	yes		yes	yes	yes	yes
__wfe			yes	yes	yes	yes	yes
__wfi			yes	yes	yes	yes	yes

Table 5-4 shows the calling syntax for each intrinsic, along with the corresponding assembly instruction and a description. See Table 5-3 for a list of which intrinsics are available on the different ARM targets. Additional intrinsices for getting and setting the CPSR register and to enable/disable interrupts are provided in Section 6.8.1.

Table 5-4. ARM Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int *count* = **__clz(**int *src* **);**	**CLZ** *count* , *src*	Returns the count of leading zeros.
void **__delay_cycles(** unsigned int *cycles* **);**	varies	Delays execution for the specified number of cycles. The number of cycles must be a constant. The __delay_cycles intrinsic inserts code to consume precisely the number of specified cycles with no side effects. The number of cycles delayed must be a compile-time constant. **Note:** Cycle timing is based on 0 wait states. Results vary with additional wait states. The implementation does not account for dynamic prediction. Lower delay cycle counts may be less accurate given pipeline flush behaviors.
unsigned int *dst* = **__get_MSP(**void **);**	**MRS** *dst* , **MSP**	Returns the current value of the Main Stack Pointer.
unsigned int *dst* = **__get_PRIMASK(**void **);**	**MRS** *dst* , **PRIMASK**	Returns the current value of the Priority Mask Register. If this value is 1, activation of all exceptions with configurable priority is prevented.
unsigned int *dest* = **__ldrex(**void* *src* **);**	**LDREX** *dst* , *src*	Loads data from memory address containing word (32-bit) data
unsigned int *dest*= **__ldrexb(**void* *src* **);**	**LDREXB** *dst* , *src*	Loads data from memory address containing byte data
unsigned long long *dest* = **__ldrexd(**void* *src* **);**	**LDREXD** *dst* , *src*	Loads data from memory address with long long support

Table 5-4. ARM Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
unsigned int *dest* = **_ _ldrexh(**void* *src* **);**	**LDREXH** *dst* , *src*	Loads data from memory address containing halfword (16-bit) data
void **__MCR (**unsigned int *coproc*, unsigned int *opc1*, unsigned int *src*, unsigned int *coproc_reg1*, unsigned int *coproc_reg2*, unsigned int *opc2* **);**	**MCR** *coproc*, *opc1*, *src*, *CR<coproc_reg1>*, *CR<coproc_reg2>*, *opc2*	Access the coprocessor registers
unsigned int **__MRC(**unsigned int *coproc*, unsigned int *opc1*, unsigned int *coproc_reg1*, unsigned int *coproc_reg2*, unsigned int *opc2* **);**	**MRC** *coproc*, *opc1*, *src*, *CR<coproc_reg1>*, *CR<coproc_reg2>*, *opc2*	Access the coprocessor registers
void **_ _nop(** void **);**	**NOP**	Perform an instruction that does nothing.
int *dst* = **_norm(**int *src* **);**	**CLZ** *dst* , *src*	Count leading zero bits. This intrinsic can be used when implementing integer normalization.
int *dst* = **_pkhbt(**int *src1* , int *src2* , int *shift* **);**	**PKHBT** *dst* , *src1* , *src2* , **#** *shift*	Combine bottom halfword of src1 with shifted top halfword of src2
nt *dst* = **_pkhtb(**int *src1* , int *src2* , int *shift* **);**	**PKHTB** *dst* , *src1* , *src2* , **#** *shift*	Combine top halfword of src1 with shifted bottom halfword of src2
int *dst* = **_qadd16(**int *src1* , int *src2* **);**	**QADD16** *dst* , *src1* , *src2*	Performs two signed halfword saturated additions
int *dst* = **_qadd8(**int *src1* , int *src2* **);**	**QADD8** *dst* , *src1* , *src2*	Performs four signed saturated 8-bit additions
int *dst* = **_qaddsubx(**int *src1* , int *src2* **);**	**QASX** *dst* , *src1* , *src2*	Exchange halfwords of src2, perform signed saturated addition on the top halfwords and signed saturated subtraction on the bottom halfwords.
int *dst* = **_qsub16(**int *src1* , int *src2* **);**	**QSUB16** *dst* , *src1* , *src2*	Performs two signed saturated halfword subtractions
int *dst* = **_qsub8(**int *src1* , int *src2* **);**	**QSUB8** *dst* , *src1* , *src2*	Performs four signed saturated 8-bit subtractions
int *dst* = **_qsubaddx(**int *src1* , int *src2* **);**	**QSAX** *dst* , *src1* , *src2*	Exchange halfwords of src2, perform signed saturated subtraction on top halfwords and signed saturated addition on bottom halfwords
int *dst* = **_ _rbit(**int *src* **);**	**RBIT** *dst* , *src*	Reverses the bit order in a word.
int *dst* = **_ _rev(**int *src* **);**	**REV** *dst* , *src*	Reverses byte order in a word. That is, converts 32-bit data between big-endian and little-endian or vice versa.
int *dst* = **_ _rev16(**int *src* **);**	**REV16** *dst* , *src*	Reverses byte order in each byte in a word independently. That is, converts 16-bit data between big-endian and little-endian or vice versa.
int *dst* = **_ _revsh(**int *src* **);**	**REVSH** *dst* , *src*	Reverses byte order in the lower byte of a word, and extends the sign to 32 bits. That is, converts 16-bit signed data to 32-bit signed data, while also converting between big-endian and little-endian or vice versa.
int *dst* = **_ _ror(**int *src* , int *shift* **);**	**ROR** *dst* , *src* , *shift*	Rotates the value to the right by the number of bits specified. Bits rotated off the right end are placed into empty bits on the left.
int *dst* =**_sadd(**int *src1* , int *src2* **);**	**QADD** *dst* , *src1* , *src2*	Saturated add
int *dst* = **_sadd16(**int *src1* , int *src2* **);**	**SADD16** *dst* , *src1* , *src2*	Performs two signed halfword additions
int *dst* = **_sadd8(**int *src1* , int *src2* **);**	**SADD8** *dst* , *src1* , *src2*	Performs four signed 8-bit additions
int *dst* = **_saddsubx(**int *src1* , int *src2* **);**	**SASX** *dst* , *src1* , *src2*	Exchange halfwords of src2, add the top halfwords and subtract the bottom halfwords
int *dst* =**_sdadd(**int *src1* , int *src2* **);**	**QDADD** *dst* , *src1* , *src2*	Saturated double-add
int *dst* =**_sdsub(**int *src1* , int *src2* **);**	**QDSUB** *dst* , *src1* , *src2*	Saturated double-subtract

Table 5-4. ARM Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int *dst* = **_sel(**int *src1* , int *src2* **);**	**SEL** *dst* , *src1* , *src2*	Selects byte n from src1 if GE bit n is set or from src2 if GE bit n is not set, where n ranges from 0 to 3.
void **_ _set_MSP(**unsigned int *src* **);**	**MSR MSP**, *src*	Sets the value of the Main Stack Pointer to src.
unsigned int *dst* = **_ _set_PRIMASK(**unsigned int *src* **);**	**MRS** *dst* , **PRIMASK** (optional) **MSR PRIMASK**, *src*	Sets the Priority Mask Register to the src value and returns the value as it was prior to being set as dst. Setting this register to 1 prevents the activation of all exceptions with configurable priority.
int *dst* = **_shadd16(**int *src1* , int *src2* **);**	**SHADD16** *dst* , *src1* , *src2*	Performs two signed halfword additions and halves the results
int *dst* = **_shadd8(**int *src1* , int *src2* **);**	**SHADD8** *dst* , *src1* , *src2*	Performs four signed 8-bit additions and halves the results
int *dst* = **_shsub16(**int *src1* , int *src2* **);**	**SHSUB16** *dst* , *src1* , *src2*	Performs two signed halfword subtractions and halves the results
int *dst* = **_shsub8**int *src1* , int *src2* **);**	**SHSUB8** *dst* , *src1* , *src2*	Performs four signed 8-bit subtractions and halves the results
int *dst* =**_smac(**int *dst* , int *src1* , int *src2* **);**	**SMULBB** *tmp* , *src1* , *src2* **QDADD** *dst* , *dst* , *tmp*	Saturated multiply-accumulate
int *dst* =**_smlabb(**int *dst* , short *src1* , short *src2* **);**	**SMLABB** *dst* , *src1* , *src2*	Signed multiply-accumulate bottom halfwords
int *dst* =**_smlabt(**int *dst* , short *src1* , int *src2* **);**	**SMLABT** *dst* , *src1* , *src2*	Signed multiply-accumulate bottom and top halfwords
int *dst* **_smlad(**int *src1* , int *src2* , int *acc* **);**	**SMLAD** *dst* , *src1* , *src2* , *acc*	Performs two signed 16-bit multiplications on the top and bottom halfwords of src1 and src2 and adds the results to acc.
int *dst* **_smladx(**int *src1* , int *src2* , int *acc* **);**	**SMLADX** *dst* , *src1* , *src2* , *acc*	Same as _smlad except the halfwords in src2 are exchange before the multiplication.
long long *dst* =**_smlalbb(**long long *dst* , short *src1* , short *src2* **);**	**SMLALBB** *dstlo* , *dsthi* , *src1* , *src2*	Signed multiply-long and accumulate bottom halfwords
long long *dst* =**_smlalbt(**long long *dst* , short *src1* , int *src2* **);**	**SMLALBT** *dstlo* , *dsthi* , *src1* , *src2*	Signed multiply-long and accumulate bottom and top halfwords
long long *dst* **_smlald(**long long *acc* , int *src1* , int *src2* **);**	**SMLALD** *dst* , *src1* , *src2*	Performs two 16-bit multiplication on the top and bottom halfwords of src1 and src2 and adds the results to the 64-bit acc operand
long long *dst* **_smlaldx(**long long *acc* , int *src1* , int *src2* **);**	**SMLALDX** *dst* , *src1* , *src2*	Same as _smlald except the halfwords in src2 are exchanged.
long long *dst* =**_smlaltb(**long long *dst* , int *src1* , short *src2* **);**	**SMLALTB** *dstlo* , *dsthi* , *src1* , *src2*	Signed multiply-long and accumulate top and bottom halfwords
long long *dst* =**_smlaltt(**long long *dst* , int *src1* , int *src2* **);**	**SMLALTT** *dstlo* , *dsthi* , *src1* , *src2*	Signed multiply-long and accumulate top halfwords
int *dst* =**_smlatb(**int *dst* , int *src1* , short *src2* **);**	**SMLATB** *dst* , *src1* , *src2*	Signed multiply-accumulate top and bottom halfwords
int *dst* =**_smlatt(**int *dst* , int *src1* , int *src2* **);**	**SMLATT** *dst* , *src1* , *src2*	Signed multiply-accumulate top halfwords
int *dst* **_smlawb(**int *src1* , short *src2* , int *acc* **);**	**SMLAWB** *dst* , *src1* , *src2*	Signed multiply-accumulate word and bottom halfword
int *dst* **_smlawt(**int *src1* , short *src2* , int *acc* **);**	**SMLAWT** *dst* , *src1* , *src2*	Signed multiply-accumulate word and top halfword
int *dst* **_smlsd(**int *src1* , int *src2* , int *acc* **);**	**SMLSD** *dst* , *src1* , *src2* , *acc*	Performs two signed 16-bit multiplications on the top and bottom halfwords of src1 and src2 and adds the difference of the results to acc.
int *dst* **_smlsdx(**int *src1* , int *src2* , int *acc* **);**	**SMLSDX** *dst* , *src1* , *src2* , *acc*	Same as _smlsd except the halfwords in src2 are exchange before the multiplication.

Table 5-4. ARM Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
long long *dst* **_smlsld(**long long *acc* , int *src1* , int *src2* **);**	**SMLSLD** *dst* , *src1* , *src2*	Performs two 16-bit multiplication on the top and bottom halfwords of src1 and src2 and adds the difference of the results to the 64-bit acc operand.
long long *dst* **_smlsldx(**long long *acc* , int *src1* , int *src2* **);**	**SMLSLDX** *dst* , *src1* , *src2*	Same as _smlsld except the halfwords in src2 are exchanged.
int *dst* **_smmla(**int *src1* , int *src2* , int *acc* **);**	**SMMLA** *dst* , *src1* , *src2* , *acc*	Performs a signed multiplication on src1 and src2, extracts the most significant 32 bits of the result, and adds an accumulate value.
int *dst* **_smmlar(**int *src1* , int *src2* , int *acc* **);**	**SMMLAR** *dst* , *src1* , *src2* , *acc*	Same as _smmla execpt the result is rounded instead of being truncated.
int *dst* **_smmls(**int *src1* , int *src2* , int *acc* **);**	**SMMLS** *dst* , *src1* , *src2* , *acc*	Performs a signed multiplication on src1 and src2, subtracts the result from an accumulate value that is shifted left by 32 bits, and extracts the most significant 32 bits of the result of the subtraction.
int *dst* **_smmlsr(**int *src1* , int *src2* , int *acc* **);**	**SMMLSR** *dst* , *src1* , *src2* , *acc*	Same as _smmls except the result is rounded instead of being truncated.
int *dst* **_smmul(**int *src1* , int *src2* , int *acc* **);**	**SMMUL** *dst* , *src1* , *src2* , *acc*	Performs a signed 32-bit multiplication on src1 and src2 and extracts the most significant 32-bits of the result.
int *dst* **_smmulr(**int *src1* , int *src2* , int *acc* **);**	**SMMULR** *dst* , *src1* , *src2* , *acc*	Same as _smmul except the result is rounded instead of being truncated.
int *dst* **=_smpy(**int *src1* , int *src2* **);**	**SMULBB** *dst* , *src1* , *src2* **QADD** *dst* , *dst* , *dst*	Saturated multiply
int *dst* **=_smsub(**int *src1* , int *src2* **);**	**SMULBB** *tmp* , *src1* , *src2* **QDSUB** *dst* , *dst* , *tmp*	Saturated multiply-subtract
int *dst* **_smuad(**int *src1* , int *src2* **);**	**SMUAD** *dst* , *src1* , *src2*	Performs two signed 16-bit multiplications on the top and bottom halfwords and adds the products.
int *dst* **_smuadx(**int *src1* , int *src2* **);**	**SMUADX** *dst* , *src1* , *src2*	Same as _smuad except the halfwords in src2 are exchange before the multiplication.
int *dst* **=_smulbb(**int *src1* , int *src2* **);**	**SMULBB** *dst* , *src1* , *src2*	Signed multiply bottom halfwords
int *dst* **=_smulbt(**int *src1* , int *src2* **);**	**SMULBT** *dst* , *src1* , *src2*	Signed multiply bottom and top halfwords
int *dst* **=_smultb(**int *src1* , int *src2* **);**	**SMULTB** *dst* , *src1* , *src2*	Signed multiply top and bottom halfwords
int *dst* **=_smultt(**int *src1* , int *src2* **);**	**SMULTT** *dst* , *src1* , *src2*	Signed multiply top halfwords
int *dst* **_smulwb(**int *src1* , short *src2* , int *acc* **);**	**SMULWB** *dst* , *src1* , *src2*	Signed multiply word and bottom halfword
int *dst* **_smulwt(**int *src1* , short *src2* , int *acc* **);**	**SMULWT** *dst* , *src1* , *src2*	Signed multiply word and top halfword
int *dst* **_smusd(**int *src1* , int *src2* **);**	**SMUSD** *dst* , *src1* , *src2*	Performs two signed 16-bit multiplications on the top and bottom halfwords and subtracts the products.
int *dst* **_smusdx(**int *src1* , int *src2* **);**	**SMUSDX** *dst* , *src1* , *src2*	Same as _smusd except the halfwords in src2 are exchanged before the multiplication.
double **__sqrt(** double **);**	**VSQRT** *dst* , *src1*	Return the square root of the specified double. This intrinsic is directly replaced with the VSQRT instruction if --fp_mode=relaxed. If strict floating point mode is used, the function must also set an errno if a domain error occurs.
float **__sqrtf(** float **);**	**VSQRT** *dst* , *src1*	Return the square root of the specified float. This intrinsic is directly replaced with the VSQRT instruction if --fp_mode=relaxed. If strict floating point mode is used, the function must also set an errno if a domain error occurs.

Table 5-4. ARM Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int *dst* =**_ssat16(**int *src* , int *bitpos* **);**	**SSAT16** *dst* , **#** *bitpos*	Performs two halfword saturations to a selectable signed range specified by bitpos
int *dst* =**_ssata(**int *src* , int *shift* , int *bitpos* **);**	**SSAT** *dst* , **#** *bitpos* , *src* , **ASR #** *shift*	Right shifts src and saturates to a selectable signed range specified by bitpos
int *dst* =**_ssatl(**int *src* , int *shift* , int *bitpos* **);**	**SSAT** *dst* , **#** *bitpos* , *src* , **LSL #** *shift*	Left shifts src and saturates to a selectable signed range specified by bitpos
int *dst* =**_ssub(**int *src1* , int *src2* **);**	**QSUB** *dst* , *src1* , *src2*	Saturated subtract
int *dst* = **_ssub16(**int *src1* , int *src2* **);**	**SSUB16** *dst* , *src1* , *src2*	Performs two signed halfword subtractions
int *dst* = **_ssub8(**int *src1* , int *src2* **);**	**SSUB8** *dst* , *src1* , *src2*	Performs four signed 8-bit subtractions
int *dst* = **_ssubaddx(**int *src1* , int *src2* **);**	**SSAX** *dst* , *src1* , *src2*	Exchange halfwords of src2, subtract the top halfwords and add the bottom halfwords
int *status* = **_ _strex(**unsigned int *src* , void* *dst* **);**	**STREX** *status* , *src* , *dest*	Stores word (32-bit) data in memory address
int *status* = **_ _strexb(**unsigned char *src* , void* *dst* **);**	**STREXB** *status* , *src* , *dest*	Stores byte data in memory address
int *status* = **_ _strexd(**unsigned long long *src* , void* *dst* **);**	**STREXD** *status* , *src* , *dest*	Stores long long data in memory address
int *status* = **_ _strexh(**unsigned short *src* , void* *dst* **);**	**STREXH** *status* , *src* , *dest*	Stores halfword (16-bit) data in memory address
int *dst* = **_subc(**int *src1* , int *src2* **);**	**SUBC** *dst* , *src1* , *src2*	Subtract with carry
int *dst* **_sxtab(**int *src1* , int *src2* , int *rotamt* **);**	**SXTAB** *dst* , *src1* , *src2* , **ROR #** *rotamt*	Extracts an optionally rotated 8-bit value from src2 and sign extends it to 32 bits, then adds the value to src1. The rotation amount can be 0, 8, 16, or 24.
int *dst* **_sxtab16(**int *src1* , int *src2* , int *rotamt* **);**	**SXTAB16** *dst* , *src1* , *src2* , **ROR #** *rotamt*	Extracts two optionally rotated 8-bit values from src2 and sign extends them to 16 bits each, then adds the values to the two 16-bit values in src1. The rotation amount should be 0, 8, 16, or 24.
int *dst* **_sxtah(**int *src1* , int *src2* , int *rotamt* **);**	**SXTAH** *dst* , *src1* , *src2* , **ROR #** *rotamt*	Extracts an optionally rotated 16-bit value from src2 and sign extends it to 32 bits, then adds the result to src1. The rotation amount can be 0, 8, 16, or 32.
int *dst* **_sxtb(**int *src1* , int *rotamt* **);**	**SXTB** *dst* , *src1* , **ROR #** *rotamt*	Extracts an optionally rotated 8-bit value from src1 and sign extends it to 32 bits. The rotation amount can be 0, 8, 16, or 24.
int *dst* **_sxtb16(**int *src1* , int *rotamt* **);**	**SXTAB16** *dst* , *src1* , **ROR #** *rotamt*	Extracts two optionally rotated 8-bit values from src1 and sign extends them to 16-bits. The rotation amount can be 0, 8, 16, or 24.
int *dst* **_sxth(**int *src1* , int *rotamt* **);**	**SXTH** *dst* , *src1* , **ROR #** *rotamt*	Extracts an optionally rotated 16-bit value from src2 and sign extends it to 32 bits. The rotation amount can be 0, 8, 16, or 24.
int *dst* = **_uadd16(**int *src1* , int *src2* **);**	**UADD16** *dst* , *src1* , *src2*	Performs two unsigned halfword additions
int *dst* = **_uadd8(**int *src1* , int *src2* **);**	**UADD8** *dst* , *src1* , *src2*	Performs four unsigned 8-bit additions
int *dst* = **_uaddsubx(**int *src1* , int *src2* **);**	**UASX** *dst* , *src1* , *src2*	Exchange halfwords of src2, add the top halfwords and subtract the bottom halfwords
int *dst* = **_uhadd16(**int *src1* , int *src2* **);**	**UHADD16** *dst* , *src1* , *src2*	Performs two unsigned halfword additions and halves the results
int *dst* = **_uhadd8(**int *src1* , int *src2* **);**	**UHADD8** *dst* , *src1* , *src2*	Performs four unsigned 8-bit additions and halves the results
int *dst* = **_uhsub16(**int *src1* , int *src2* **);**	**UHSUB16** *dst* , *src1* , *src2*	Performs two unsigned halfword subtractions and halves the results
int *dst* = **_uhsub8(**int *src1* , int *src2* **);**	**UHSUB8** *dst* , *src1* , *src2*	Performs four unsigned 8-bit subtractions and halves the results

Table 5-4. ARM Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
int *dst* = **_umaal(**long long *acc* , int *src1* , int *src2* **);**	**UMAAL** *dst1* , *dst2* , *src1* , *src2*	Performs an unsigned 32-bit multiplication on src1 and src2, then adds two unsigned 32-bit values in acc.
int *dst* = **_uqadd16(**int *src1* , int *src2* **);**	**UQADD16** *dst* , *src1* , *src2*	Performs two unsigned halfword saturated additions
int *dst* = **_uqadd8(**int *src1* , int *src2* **);**	**UQADD8** *dst* , *src1* , *src2*	Performs four unsigned saturated 8-bit additions
int *dst* = **_uqaddsubx(**int *src1* , int *src2* **);**	**UQASX** *dst* , *src1* , *src2*	Exchange halfwords of src2, perform unsigned saturated addition on the top halfwords and unsigned saturated subtraction on the bottom halfwords.
int *dst* = **_uqsub16(**int *src1* , int *src2* **);**	**UQSUB16** *dst* , *src1* , *src2*	Performs two unsigned saturated halfword subtractions
int *dst* = **_uqsub8(**int *src1* , int *src2* **);**	**UQSUB8** *dst* , *src1* , *src2*	Performs four unsigned saturated 8-bit subtractions
int *dst* = **_uqsubaddx(**int *src1* , int *src2* **);**	**UQSAX** *dst* , *src1* , *src2*	Exchange halfwords of src2, perform unsigned saturated subtraction on top halfwords and unsigned saturated addition on bottom halfwords
int *dst* = **_usad8(**int *src1* , int *src2* **);**	**USAD8** *dst* , *src1* , *src2*	Performs four unsigned 8-bit subtractions, and adds the absolute value of the differences together.
int *dst* =**_usat16(**int *src* , int *bitpos* **);**	**USAT16** *dst* , **#** *bitpos*	Performs two halfword saturations to a selectable unsigned range specified by bitpos
int *dst* =**_usata(**int *src* , int *shift* , int *bitpos* **);**	**USAT** *dst* , **#** *bitpos* , *src* , **ASR #** *shift*	Right shifts src and saturates to a selectable unsigned range specified by bitpos
int *dst* =**_usatl(**int *src* , int *shift* , int *bitpos* **);**	**USAT** *dst* , **#** *bitpos* , *src* , **LSL #** *shift*	Left shifts src and saturates to a selectable unsigned range specified by bitpos
int *dst* = **_usub16(**int *src1* , int *src2* **);**	**USUB16** *dst* , *src1* , *src2*	Performs two unsigned halfword subtractions
int *dst* = **_usub8(**int *src1* , int *src2* **);**	**USUB8** *dst* , *src1* , *src2*	Performs four unsigned 8-bit subtractions
int *dst* = **_usubaddx(**int *src1* , int *src2* **);**	**USAX** *dst* , *src1* , *src2*	Exchange halfwords of src2, subtract the top halfwords and add the bottom halfwords
int *dst* **_uxtab(**int *src1* , int *src2* , int *rotamt* **);**	**UXTAB** *dst* , *src1* , *src2* , **ROR #** *rotamt*	Extracts an optionally rotated 8-bit value from src2 and zero extends it to 32 bits, then adds the value to src1. The rotation amount can be 0, 8, 16, or 24.
int *dst* **_uxtab16(**int *src1* , int *src2* , int *rotamt* **);**	**UXTAB16** *dst* , *src1* , *src2* , **ROR #** *rotamt*	Extracts two optionally rotated 8-bit values from src2 and zero extends them to 16 bits each, then adds the values to the two 16-bit values in src1. The rotation amount should be 0, 8, 16, or 24.
int *dst* **_uxtah(**int *src1* , int *src2* , int *rotamt* **);**	**UXTAH** *dst* , *src1* , *src2* , **ROR #** *rotamt*	Extracts an optionally rotated 16-bit value from src2 and zero extends it to 32 bits, then adds the result to src1. The rotation amount can be 0, 8, 16, or 32.
int *dst* **_uxtb(**int *src1* , int *rotamt* **);**	**UXTB** *dst* , *src1* , **ROR #** *rotamt*	Extracts an optionally rotated 8-bit value from src2 and zero extends it to 32 bits. The rotation amount can be 0, 8, 16, or 24.
int *dst* **_uxtb16(**int *src1* , int *rotamt* **);**	**UXTB16** *dst* , *src1* , **ROR #** *rotamt*	Extracts two optionally rotated 8-bit values from src1 and zero extends them to 16-bits. The rotation amount can be 0, 8, 16, or 24.
int *dst* **_uxth(**int *src1* , int *rotamt* **);**	**UXTH** *dst* , *src1* , **ROR #** *rotamt*	Extracts an optionally rotated 16-bit value from src2 and zero extends it to 32 bits. The rotation amount can be 0, 8, 16, or 24.

Table 5-4. ARM Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
void **_ _wfe(** void **);**	**WFE**	Wait for event. Save power by waiting for an exception or event..
void **_ _wfi(** void **);**	**WFI**	Wait for interrupt. Enter standby, dormant or shutdown mode, where an interrupt is required to wake-up the processor.

In addition, the compiler supports many of the intrinsics described in the ARM C Language Extensions (ACLE) specification. These intrinsics are applicable for the Cortex-M and Cortex-R processor variants. The ACLE intrinsics are implemented in order to support the development of source code that can be compiled using ACLE-compliant compilers from multiple vendors for a variety of ARM processors. A number of the intrinsics are duplicates of intrinsics listed in the previous table but with slightly different names (such as one vs. two leading underscores).

The compiler does not support all of the ACLE intrinsics listed in the ACLE specification. For example, the __cls, __clsl, and __clsll ACLE intrinsics are not supported, because the CLS instruction is not available on the Cortex-M or Cortex-R architectures.

In order to use the ACLE intrinsics, your code must include the provided `arm_acle.h` header file. For details about the ACLE intrinsics, see the ACLE specification. For information about which ACLE intrinsics are supported, see the `arm_acle.h` file. Where applicable, the declarations of ACLE intrinsics that are not supported are enclosed in comments in that header file along with a brief explanation of why the intrinsic is not supported and a reference to the appropriate section in the ACLE specification where the intrinsic is described.

5.15 Object File Symbol Naming Conventions (Linknames)

Each externally visible identifier is assigned a unique symbol name to be used in the object file, a so-called *linkname*. This name is assigned by the compiler according to an algorithm which depends on the name, type, and source language of the symbol. This algorithm may add a prefix to the identifier (typically an underscore), and it may *mangle* the name.

User-defined symbols in C code and in assembly code are stored in the same namespace, which means you are responsible for making sure that your C identifiers do not collide with your assembly code identifiers. You may have identifiers that collide with assembly keywords (for instance, register names); in this case, the compiler automatically uses an escape sequence to prevent the collision. The compiler escapes the identifier with double parallel bars, which instructs the assembler not to treat the identifier as a keyword. You are responsible for making sure that C identifiers do not collide with user-defined assembly code identifiers.

Name mangling encodes the types of the parameters of a function in the linkname for a function. Name mangling only occurs for C++ functions which are not declared 'extern "C"'. Mangling allows function overloading, operator overloading, and type-safe linking. Be aware that the return value of the function is not encoded in the mangled name, as C++ functions cannot be overloaded based on the return value.

For example, the general form of a C++ linkname for a 32-bit 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 compiled in 16-bit mode
```

This is the resulting assembly code:

```
$__foo__Fi
```

The linkname of foo is $__foo__Fi, indicating that foo is a 16-bit 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.

The mangling algorithm follows that described in the Itanium C++ ABI (http://www.codesourcery.com/cxx-abi/abi.html).

`int foo(int i) { }` would be mangled "_Z3fooi"

Note

EABI Mode C++ Demangling: The EABI mode has a different C++ demangling scheme. For instance, there is no prefix (either _ or $). Please refer to the ARM Information Center for details.

5.16 Changing the ANSI/ISO C/C++ Language Mode

The language mode command-line options determine how the compiler interprets your source code. You specify one option to identify which language standard your code follows. You can also specify a separate option to specify how strictly the compiler should expect your code to conform to the standard.

Specify one of the following language options to control the language standard that the compiler expects the source to follow. The options are:

- ANSI/ISO C89 (--c89, default for C files)
- ANSI/ISO C99 (--c99, see Section 5.16.1.)
- ANSI/ISO C11 (--c11, see Section 5.16.2)
- ISO C++14 (--c++14 , used for all C++ files , see Section 5.2.)

Use one of the following options to specify how strictly the code conforms to the standard:

- Relaxed ANSI/ISO (--relaxed_ansi or -pr) This is the default.
- Strict ANSI/ISO (--strict_ansi or -ps)

The default is relaxed ANSI/ISO mode. Under relaxed ANSI/ISO mode, the compiler accepts language extensions that could potentially conflict with ANSI/ISO C/C++. Under strict ANSI mode, these language extensions are suppressed so that the compiler will accept all strictly conforming programs. (See Section 5.16.3.)

If you want to link object files created with the TI CodeGen tools with object files generated by other compiler tool chains, the ARM standard specifies that you should define the _AEABI_PORTABILITY_LEVEL preprocessor symbol as follows before #including any standard header files, such as <stdlib.h>.

```
#define _AEABI_PORTABILITY_LEVEL 1
```

This definition enables full portability. Defining the symbol to 0 specifies that the "C standard" portability level will be used.

5.16.1 C99 Support (--c99)

The compiler supports the 1999 standard of C as standardized by the ISO. However, the following list of run-time functions and features are *not* implemented or fully supported:

- inttypes.h
  - wcstoimax() / wcstoumax()
- stdio.h
  - The %e specifier may produce "-0" when "0" is expected by the standard
  - snprintf() does not properly pad with spaces when writing to a wide character array
- stdlib.h
  - vfscanf() / vscanf() / vsscanf() return value on floating point matching failure is incorrect
- wchar.h
  - getws() / fputws()

- mbrlen()
- mbsrtowcs()
- wcscat()
- wcschr()
- wcscmp() / wcsncmp()
- wcscpy() / wcsncpy()
- wcsftime()
- wcsrtombs()
- wcsstr()
- wcstok()
- wcsxfrm()
- Wide character print / scan functions
- Wide character conversion functions

5.16.2 C11 Support (--c11)

The compiler supports the 2011 standard of C as standardized by the ISO. However, in addition to the list in Section 5.16.1, the following run-time functions and features are *not* implemented or fully supported in C11 mode:

- threads.h

5.16.3 Strict ANSI Mode and Relaxed ANSI Mode (--strict_ansi and --relaxed_ansi)

Under relaxed ANSI/ISO mode (the default), the compiler accepts language extensions that could potentially conflict with a strictly conforming ANSI/ISO C/C++ program. Under strict ANSI mode, these language extensions are suppressed so that the compiler will accept all strictly conforming programs.

Use the --strict_ansi option when you know your program is a conforming program and it will not compile in relaxed mode. In this mode, language extensions that conflict with ANSI/ISO C/C++ are disabled and the compiler will emit error messages where the standard requires it to do so. Violations that are considered discretionary by the standard may be emitted as warnings instead.

Examples:

The following is strictly conforming C code, but will not be accepted by the compiler in the default relaxed mode. To get the compiler to accept this code, use strict ANSI mode. The compiler will suppress the interrupt keyword language exception, and interrupt may then be used as an identifier in the code.

```
int main()
{
    int interrupt = 0;
    return 0;
}
```

The following is not strictly conforming code. The compiler will not accept this code in strict ANSI mode. To get the compiler to accept it, use relaxed ANSI mode. The compiler will provide the interrupt keyword extension and will accept the code.

```
interrupt void isr(void);
int main()
{
    return 0;
}
```

The following code is accepted in all modes. The __interrupt keyword does not conflict with the ANSI/ISO C standard, so it is always available as a language extension.

```
__interrupt void isr(void);
int main()
{
    return 0;
}
```

The default mode is relaxed ANSI. This mode can be selected with the --relaxed_ansi (or -pr) option. Relaxed ANSI mode accepts the broadest range of programs. It accepts all TI language extensions, even those which conflict with ANSI/ISO, and ignores some ANSI/ISO violations for which the compiler can do something reasonable. Some GCC language extensions described in Section 5.17 may conflict with strict ANSI/ISO standards, but many do not conflict with the standards.

5.17 GNU , Clang, and ACLE Language Extensions

The GNU compiler collection (GCC) defines a number of language features not found in the ANSI/ISO C and C++ standards. The definition and examples of these extensions (for GCC version 4.7) can be found at the GNU web site, http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/C-Extensions.html#C-Extensions. Most of these extensions are also available for C++ source code.

The compiler also supports the following Clang macro extensions, which are described in the Clang 6 Documentation:

- __has_feature (up to tests described for Clang 3.5)
- __has_extension (up to tests described for Clang 3.5)
- __has_include
- __has_include_next
- __has_builtin (see Section 5.17.6)
- __has_attribute

In addition, the compiler supports many of the features described in the ARM C Language Extensions (ACLE) specification. These features are applicable for the Cortex-M and Cortex-R processor variants. ACLE support affects the pre-defined macros (Table 2-31), function attributes (Section 5.17.2), and intrinsics (Section 5.14) you may use in C/C++ code. These features are implemented in order to support the development of source code that can be compiled using ACLE-compliant compilers from multiple vendors for a variety of ARM processors.

5.17.1 Extensions

Most of the GCC language extensions are available in the TI compiler when compiling in relaxed ANSI mode (--relaxed_ansi).

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

Table 5-5. 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

Table 5-5. GCC Language Extensions (continued)

Extensions	Descriptions
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.
Dollar signs	A dollar sign is allowed in identifiers.
Character escapes	The character ESC is represented as \e
Variable attributes	Specifying the attributes of variables
Type attributes	Specifying the attributes of types
Alignment	Inquiring about the alignment of a type or variable
Inline	Defining inline functions (as fast as macros)
Assembly labels	Specifying the assembler name to use for a C symbol
Extended asm	Assembler instructions with C operands
Constraints	Constraints for asm operands
Wrapper headers	Wrapper header files can include another version of the header file using #include_next
Alternate keywords	Header files can use __const__, __asm__, etc
Explicit reg vars	Defining variables residing in specified registers
Incomplete enum types	Define an enum tag without specifying its possible values
Function names	Printable strings which are the name of the current function
Return address	Getting the return or frame address of a function (limited support)
Other built-ins	Other built-in functions (see Section 5.17.6)
Vector extensions	Using vector instructions through built-in functions
Target built-ins	Built-in functions specific to particular targets
Pragmas	Pragmas accepted by GCC
Unnamed fields	Unnamed struct/union fields within structs/unions
Thread-local	Per-thread variables
Binary constants	Binary constants using the '0b' prefix.

(1) Feature defined for GCC 3.0; definition and examples at http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/C-Extensions.html#C-Extensions

5.17.2 Function Attributes

The following GCC function attributes are supported:

- alias
- aligned
- always_inline
- calls
- const
- constructor
- deprecated
- format
- format_arg
- interrupt
- malloc
- naked
- noinline
- noreturn
- pure
- section
- target
- unused
- used
- warn_unused_result
- weak

The following additional TI-specific function attributes are supported:

- retain
- ramfunc

For example, this function declaration uses the **alias** attribute to make "my_alias" a function alias for the "myFunc" function:

```
void my_alias() __attribute__((alias("myFunc")));
```

The **aligned** function attribute aligns the function using the specified alignment. The alignment must be a power of 2.

The **always_inline** function attribute has the same effect as the FUNC_ALWAYS_INLINE pragma. See Section 5.11.12

The **calls** attribute has the same effect as the CALLS pragma, which is described in Section 5.11.1.

The **format** attribute is applied to the declarations of printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf, scanf, fscanf, vfscanf, vscanf, vsscanf, and sscanf in stdio.h. Thus when GCC extensions are enabled, the data arguments of these functions are type checked against the format specifiers in the format string argument and warnings are issued when there is a mismatch. These warnings can be suppressed in the usual ways if they are not desired.

See Section 5.11.16 for more about using the **interrupt** function attribute.

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

The **naked** attribute identifies functions written as embedded assembly functions using `__asm` statements. The compiler does not generate prologue and epilog sequences for such functions. See Section 5.10.

The **noinline** function attribute has the same effect as the FUNC_CANNOT_INLINE pragma. See Section 5.11.13

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

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

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

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

The **target** attribute causes a function to be compiled in either ARM (32-bit) or Thumb (16-bit) mode. The target attribute has the same effect as the CODE_STATE pragma. The following examples use the target attribute.

```
__attribute__((target("arm"))) void foo(int arg1, int arg2)
__attribute__((target("thumb"))) void foo(int arg1, int arg2)
```

Note that the "pcs" attribute described in the ACLE specification is not supported.

The **retain** attribute has the same effect as the RETAIN pragma (Section 5.11.27). That is, the section that contains the function will not be omitted from conditionally linked output even if it is not referenced elsewhere in the application.

The **section** attribute when used on a function has the same effect as the CODE_SECTION pragma. See Section 5.11.4

The **weak** attribute has the same effect as the WEAK pragma (Section 5.11.32).

5.17.3 For Loop Attributes

If you are using C++, there are several TI-specific attributes that can be applied to loops. No corresponding syntax is available in C. The following TI-specific attributes have the same function as their corresponding pragmas:

- TI::must_iterate
- TI::unroll

See Section 5.11.18.1 for an example that uses a for loop attribute.

5.17.4 Variable Attributes

The following variable attributes are supported:

- aligned
- deprecated
- location
- mode
- noinit
- packed
- persistent
- retain
- section
- transparent_union
- unused
- used
- weak

The **aligned** attribute used on a variable has the same effect as the DATA_ALIGN pragma. See Section 5.11.6

The **location** attribute has the same effect as the LOCATION pragma. See Section 5.11.17. For example:

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

The **noinit** and **persistent** attributes apply to the ROM initialization model and allow an application to avoid initializing certain global variables during a reset. The alternative RAM initialization model initializes variables only when the image is loaded; no variables are initialized during a reset. See the "RAM Model vs. ROM Model" section and its subsections in the *ARM Assembly Language Tools User's Guide*.

The **noinit** attribute can be used on uninitialized variables; it prevents those variables from being set to 0 during a reset. The **persistent** attribute can be used on initialized variables; it prevents those variables from being initialized during a reset. By default, variables marked noinit or persistent will be placed in sections named `.TI.noinit` and `.TI.persistent` , respectively. The location of these sections is controlled by the linker command file. Typically .TI.persistent sections are placed in FRAM for devices that support FRAM and .TI.noinit sections are placed in RAM. Also see Section 5.11.19.

The **packed** attribute may be applied to individual fields within a struct or union. The packed attribute is supported on all ARM targets. See the description of the --unaligned_access option for more information on how the compiler accesses unaligned data.

The **retain** attribute has the same effect as the RETAIN pragma (Section 5.11.27). That is, the section that contains the variable will not be omitted from conditionally linked output even if it is not referenced elsewhere in the application.

The **section** attribute when used on a variable has the same effect as the DATA_SECTION pragma. See Section 5.11.7

The **used** attribute is defined in GCC 4.2 (see http://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/Variable-Attributes.html#Variable-Attributes).

The **weak** attribute has the same effect as the WEAK pragma (Section 5.11.32).

5.17.5 Type Attributes

The following type attributes are supported:

- aligned
- deprecated
- packed
- transparent_union
- unused

The **packed** attribute is supported for struct and union types. It is supported on all ARM targets if the --relaxed_ansi option is used. See the description of the --unaligned_access option for more information on how the compiler accesses unaligned data.

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

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

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

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

```
void foo(int *param);
struct packed_struct ps;
int *p1 = &ps.i;
int *p2 = (int *)&ps.i;
foo(&ps.i);
```

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

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

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

5.17.6 Built-In Functions

The following built-in functions are supported:

- __builtin_abs()
- __builtin_constant_p()
- __builtin_expect()
- __builtin_fabs()
- __builtin_fabsf()
- __builtin_frame_address()
- __builtin_labs()
- __builtin_llabs()
- __builtin_sqrt()
- __builtin_sqrtf()
- __builtin_memcpy()
- __builtin_return_address()

The __builtin_frame_address() function always returns zero unless the argument is a constant zero.

The __builtin_sqrt() and __builtin_sqrtf() functions are supported only when hardware floating point support is enabled. In addition, the __builtin_sqrt() function is not supported if --float_support is set to fpv4spd16.

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

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

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

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

The __builtin_return_address() function always returns zero.

5.18 AUTOSAR

The ARM compiler supports the AUTOSAR 3.1 standard by providing the following header files:

- Compiler.h
- Platform_Types.h
- Std_Types.h
- Compiler_Cfg.h

Compiler_Cfg.h is an empty file, the contents of which should be provided by the end user. The provided file contains information on what the contents of the file should look like. It is included by Compiler.h. If a new Compiler_Cfg.h file is provided by the user, its include path must come before the path to the run-time-support header files.

More information on AUTOSAR can be found at http://www.autosar.org.

5.19 Compiler Limits

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

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

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

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

Chapter 6

Run-Time Environment

This chapter describes the ARM 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.

6.1 Memory Model 138
6.2 Object Representation 140
6.3 Register Conventions 147
6.4 Function Structure and Calling Conventions 149
6.5 Accessing Linker Symbols in C and C++ 152
6.6 Interfacing C and C++ With Assembly Language 152
6.7 Interrupt Handling 155
6.8 Intrinsic Run-Time-Support Arithmetic and Conversion Routines 159
6.9 Built-In Functions 160
6.10 System Initialization 160
6.11 Dual-State Interworking Under TIABI (Deprecated) 169

6.1 Memory Model

The ARM compiler treats memory as a single linear block that is partitioned into subblocks of code and data. Each subblock of code or data generated by a C program is placed in its own continuous memory space. The compiler assumes that a full 32-bit address space is available in target memory.

Note

The Linker Defines the Memory Map

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

6.1.1 Sections

The compiler produces relocatable blocks of code and data called *sections*, which are allocated in memory in a variety of ways to conform to a various system configurations. For information about sections and allocating them, see the introductory object file information in the *ARM Assembly Language Tools User's Guide*.

There are two basic types of sections:

- **Initialized sections** contain data or executable code. Initialized sections are usually read-only; exceptions are noted below. The C/C++ compiler creates the following initialized sections:
  - The **.binit section** contains boot time copy tables. For details on BINIT, see the *ARM Assembly Language Tools User's Guide*.
  - The **.init_array section** contains global constructor tables.
  - The **.ovly section** contains copy tables for unions in which different sections have the same run address.
  - The **.data section** contains initialized global and static variables. This section is writable.
  - The **.const section** contains read-only data, typically string constants and static-scoped objects defined with the C/C++ qualifier *const*. Note that not all static-scoped objects marked "const" are placed in the .const section (see Section 5.7.1).
  - The **.text section** contains all the executable code. It also contains string literals, switch tables, and compiler-generated constants. This section is usually read-only. Note that some string literals may instead be placed in .const:.string. The placement of string literals depends on the size of the string and the use of the --embedded_constants option.
  - The **.TI.crctab section** contains CRC checking tables.
- **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at run time to create and store variables. The compiler creates the following uninitialized sections:
  - For EABI only, the **.bss section** reserves space for uninitialized global and static variables. Uninitialized variables that are also unused are usually created as common symbols (unless you specify --common=off) instead of being placed in .bss so that they can be excluded from the resulting application.
  - The **.stack section** reserves memory for the C/C++ software stack.
  - The **.sysmem section** reserves space for dynamic memory allocation. This space is used by dynamic memory allocation routines, such as malloc(), calloc(), realloc(), or new(). If a C/C++ program does not use these functions, the compiler does not create the .sysmem section.

The assembler creates the default sections .text, .bss, and .data. You can instruct the compiler to create additional sections by using the CODE_SECTION and DATA_SECTION pragmas (see Section 5.11.4 and Section 5.11.7).

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
.bss	RAM	.pinit	ROM or RAM
.cinit	ROM or RAM	.stack	RAM
.const	ROM or RAM	.sysmem	RAM
.data	RAM	.text	ROM or RAM
.init_array	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 *ARM Assembly Language Tools User's Guide*.

6.1.2 C/C++ System Stack

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

- Allocate local variables
- Pass arguments to functions
- Save register contents

The run-time stack grows from the high addresses to the low addresses. The compiler uses the R13 register to manage this stack. R13 is the *stack pointer* (SP), which points to the next unused location on the stack.

The linker sets the stack size, creates a global symbol, __TI_STACK_SIZE, and assigns it a value equal to the stack size in bytes. The default stack size is 2048 bytes. You can change the stack size at link time by using the --stack_size option with the linker command. For more information on the --stack_size option, see the linker description chapter in the *ARM Assembly Language Tools User's Guide*.

At system initialization, SP is set to a designated address for the top of the stack. This address is the first location past the end of the .stack section. Since the position of the stack depends on where the .stack section is allocated, the actual address of the stack is determined at link time.

The C/C++ environment automatically decrements SP at the entry to a function to reserve all the space necessary for the execution of that function. The stack pointer is incremented at the exit of the function to restore the stack to the state before the function was entered. If you interface assembly language routines to C/C++ programs, be sure to restore the stack pointer to the same state it was in before the function was entered.

For more information about using the stack pointer, see Section 6.3; for more information about the stack, see Section 6.4.

Note

Stack Overflow: The compiler provides no means to check for stack overflow during compilation or at run time. A stack overflow disrupts the run-time environment, causing your program to fail. Be sure to allow enough space for the stack to grow. You can use the --entry_hook option to add code to the beginning of each function to check for stack overflow; see Section 2.15.

6.1.3 Dynamic Memory Allocation

The run-time-support library supplied with the ARM compiler contains several functions (such as malloc, calloc, and realloc) that allow you to allocate memory dynamically for variables at run time.

Memory is allocated from a global pool, or heap, that is defined in the .sysmem section. You can set the size of the .sysmem section by using the --heap_size=*size* option with the linker command. The linker also creates a global symbol, __TI_SYSMEM_SIZE, and assigns it a value equal to the size of the heap in bytes. The default size is 2048 bytes. For more information on the --heap_size option, see the linker description chapter in the *ARM Assembly Language Tools User's Guide*.

If you use any C I/O function, the RTS library allocates an I/O buffer for each file you access. This buffer will be a bit larger than BUFSIZ, which is defined in stdio.h and defaults to 256. Make sure you allocate a heap large enough for these buffers or use setvbuf to change the buffer to a statically-allocated buffer.

Dynamically allocated objects are not addressed directly (they are always accessed with pointers) and the memory pool is in a separate section (.sysmem). Therefore, the dynamic memory pool size may be limited only by the amount of memory in your system. To conserve space in the .bss section, you can allocate large arrays from the heap instead of defining them as global or static. For example, instead of a definition such as:

```
struct big table[100];
```

Use a pointer and call the malloc function:

```
struct big *table
table = (struct big *)malloc(100*sizeof(struct big));
```

When allocating from a heap, make sure the size of the heap is large enough for the allocation. This is particularly important when allocating variable-length arrays.

6.2 Object Representation

For general information about data types, see Section 5.5. This section explains how various data objects are sized, aligned, and accessed.

6.2.1 Data Type Storage

Table 6-2 lists register and memory storage for various data types:

Table 6-2. Data Representation in Registers and Memory

Data Type	Register Storage	Memory Storage
char, signed char	Bits 0-7 of register(1)	8 bits aligned to 8-bit boundary
unsigned char, bool	Bits 0-7 of register	8 bits aligned to 8-bit boundary
short, signed short	Bits 0-15 of register(1)	16 bits aligned to 16-bit (halfword) boundary
unsigned short, wchar_t	Bits 0-15 of register	16 bits aligned to 16-bit (halfword) boundary
int, signed int	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
unsigned int	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
long, signed long	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
unsigned long	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
long long	Even/odd register pair	64 bits aligned to 32-bit (word) boundary(2)
unsigned long long	Even/odd register pair	64 bits aligned to 32-bit (word) boundary(2)
float	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
double	Register pair	64 bits aligned to 32-bit (word) boundary(2)
long double	Register pair	64 bits aligned to 32-bit (word) boundary(2)
struct	Members stored as individual types require.	Members are stored as their individual types require; aligned according to the member with the most restrictive alignment requirement.
array	Members stored as individual types require.	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	Bits 0-31 of register	32 bits aligned to 32-bit (word) boundary
pointer to member function	Components stored as individual types require	64 bits aligned to 32-bit (word) boundary

(1) Negative values are sign-extended to bit 31.
(2) 64-bit data is aligned on a 64-bit boundary.

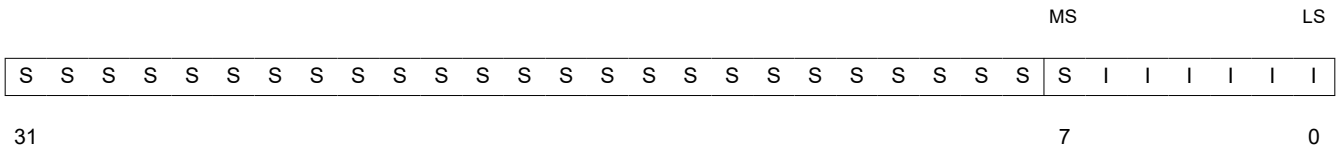
For details about the size of an enum type, see Table 5-2.

6.2.1.1 char and short Data Types (signed and unsigned)

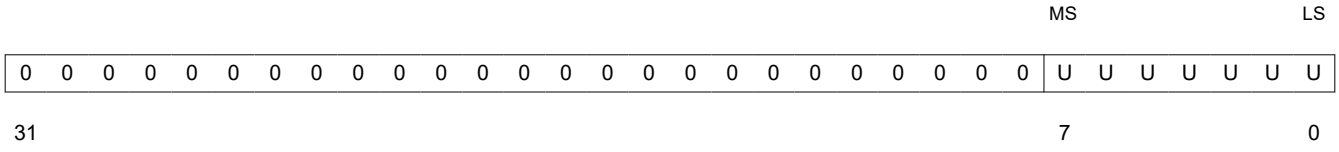
The char and unsigned char data types are stored in memory as a single byte and are loaded to and stored from bits 0-7 of a register (see Figure 6-1). Objects defined as short or unsigned short are stored in memory as two bytes at a halfword (2 byte) aligned address and they are loaded to and stored from bits 0-15 of a register (see Figure 6-1).

Figure 6-1. Char and Short Data Storage Format

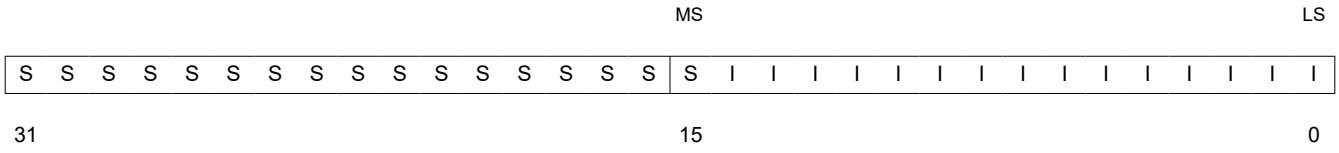
Signed 8-bit char



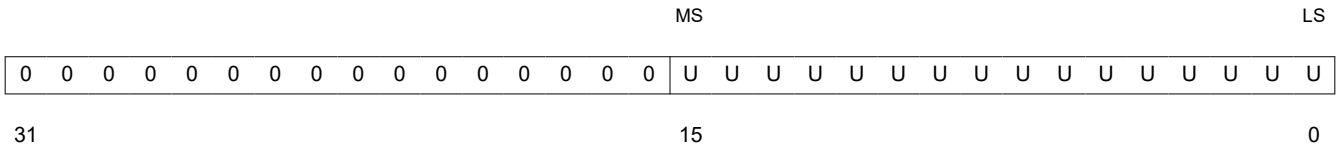
Unsigned 8-bit char



Signed 16-bit short



Unsigned 16-bit short

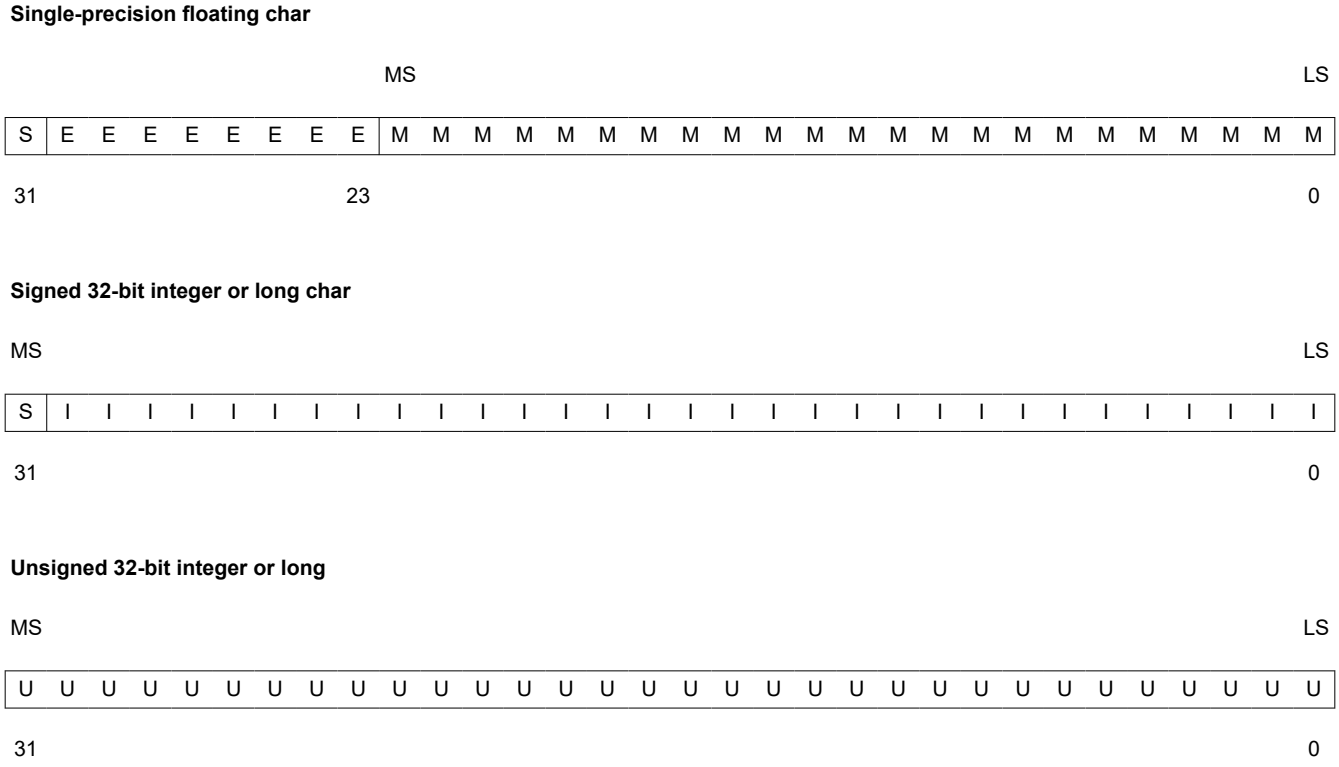


LEGEND: S = sign, I = signed integer, U = unsigned integer, MS = most significant, LS = least significant

6.2.1.2 float, int, and long Data Types (signed and unsigned)

The int, unsigned int, float, long and unsigned long data types are stored in memory as 32-bit objects at word (4 byte) aligned addresses. Objects of these types are loaded to and stored from bits 0-31 of a register, as shown in Figure 6-2. In big-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 24-31 of the register, moving the second byte of memory to bits 16-23, moving the third byte to bits 8-15, and moving the fourth byte to bits 0-7. In little-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 0-7 of the register, moving the second byte to bits 8-15, moving the third byte to bits 16-23, and moving the fourth byte to bits 24-31.

Figure 6-2. 32-Bit Data Storage Format



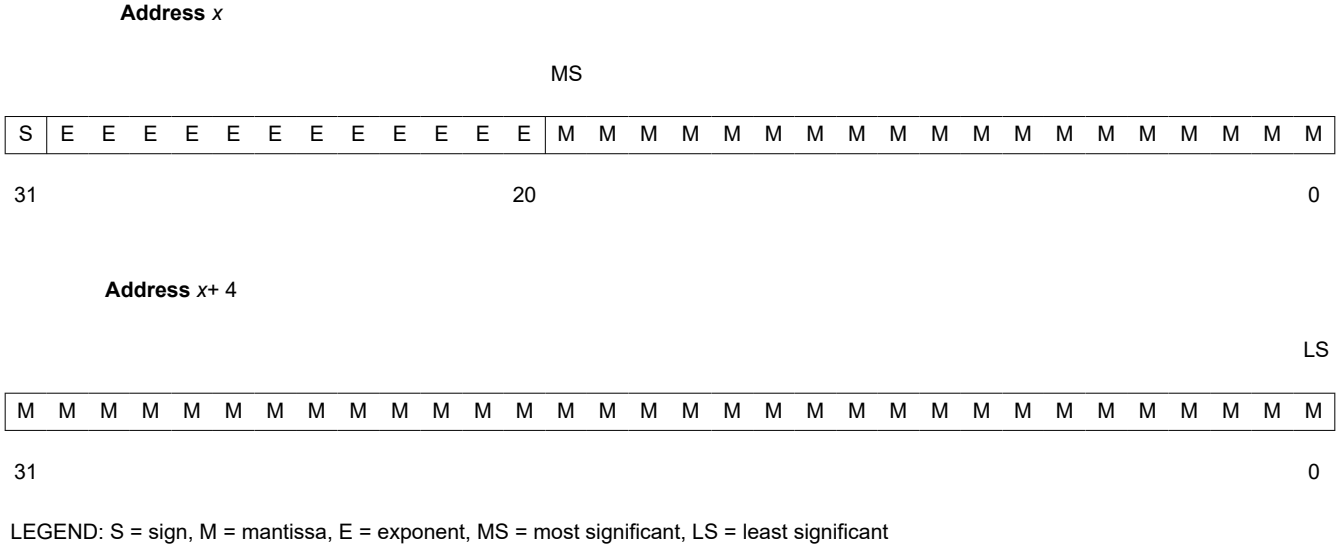
LEGEND: S = sign, M = Mantissa, U = unsigned integer, E = exponent, I = signed integer, MS = most significant, LS = least significant

6.2.1.3 double, long double, and long long Data Types (signed and unsigned)

Double, long double, long long and unsigned long long data types are stored in memory in a pair of registers and are always referenced as a pair. These types are stored as 64-bit objects at word (4 byte) aligned addresses. For FPA mode, the word at the lowest address contains the sign bit, the exponent, and the most significant part of the mantissa. The word at the higher address contains the least significant part of the mantissa. This is true regardless of the endianness of the target. For VFP mode, the words are ordered based upon the endianness of the target.

Objects of this type are loaded into and stored in register pairs, as shown in the following figure. The most significant memory word contains the sign bit, exponent, and the most significant part of the mantissa. The least significant memory word contains the least significant part of the mantissa.

Figure 6-3. Double-Precision Floating-Point Data Storage Format



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

6.2.1.4 Pointer to Data Member Types

Pointer to data member objects are stored in memory like an unsigned int (32 bit) integral type. Its value is the byte offset to the data member in the class, plus 1. The zero value is reserved to represent the NULL pointer to the data member.

6.2.1.5 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 non-virtual. The parameter f is the pointer to the member function if it is non-virtual, when I is 0. The 0 is the offset to the virtual function pointer within the class object.

6.2.1.6 Structure and Array Alignment

Structures are aligned according to the member with the most restrictive alignment requirement. Structures are padded so that the size of the structure is a multiple of its alignment. Arrays are always word aligned. Elements of arrays are stored in the same manner as if they were individual objects.

6.2.2 Bit Fields

Bit fields are the only objects that are packed within a byte. That is, two bit fields can be stored in the same byte. Bit fields can range in size from 1 to 32 bits, but they never span a 4-byte boundary.

For big-endian mode, bit fields are packed into registers from most significant bit (MSB) to least significant bit (LSB) in the order in which they are defined. Bit fields are packed in memory from most significant byte (MSbyte) to least significant byte (LSbyte). For little-endian mode, bit fields are packed into registers from the LSB to the MSB in the order in which they are defined, and packed in memory from LSbyte to MSbyte.

Here are some details about how bit fields are handled:

- Plain int bit fields are unsigned. Consider the following C code:

```
struct st
{
   int a:5;
} S;
foo()
{
   if (S.a < 0)
      bar();
}
```

  In this example, bar () is never called as bit field 'a' is unsigned. Use signed int if you need a signed bit field.
- Bit fields of type long long are supported.
- Bit fields are treated as the declared type.
- The size and alignment of the struct containing the bit field depends on the declared type of the bit field. For example, consider the struct:

```
struct st {int a:4};
```

  This struct uses up 4 bytes and is aligned at 4 bytes.
- Unnamed bit fields affect the alignment of the struct or union. For example, consider the struct:

```
struct st{char a:4; int :22;};
```

  This struct uses 4 bytes and is aligned at a 4-byte boundary.
- Bit fields declared volatile are accessed according to the bit field's declared type. A volatile bit field reference generates exactly one reference to its storage; multiple volatile bit field accesses are not merged.

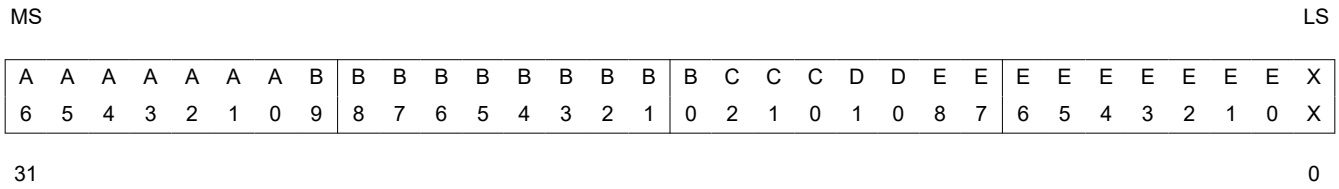
Figure 6-4 illustrates bit-field packing, using the following bit field definitions:

```
struct{
int A:7
int B:10
int C:3
int D:2
int E:9
}x;
```

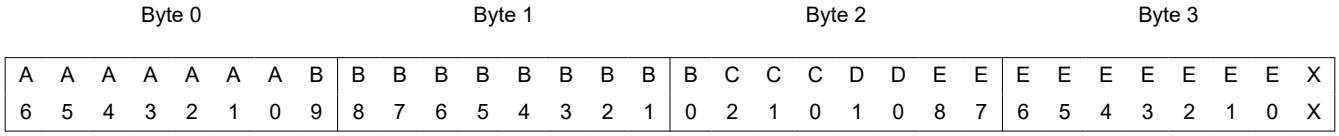
A0 represents the least significant bit of the field A; A1 represents the next least significant bit, etc. Again, storage of bit fields in memory is done with a byte-by-byte, rather than bit-by-bit, transfer.

Figure 6-4. Bit-Field Packing in Big-Endian and Little-Endian Formats

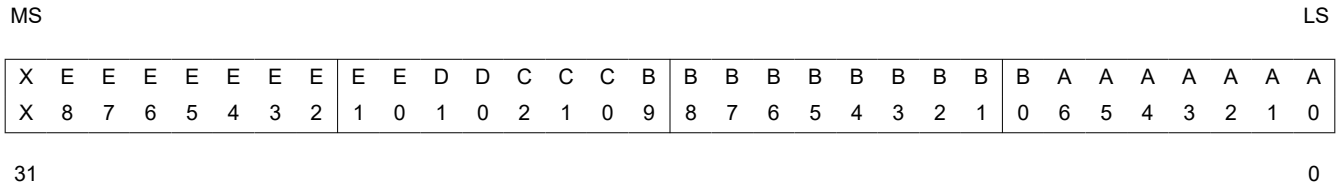
Big-endian register



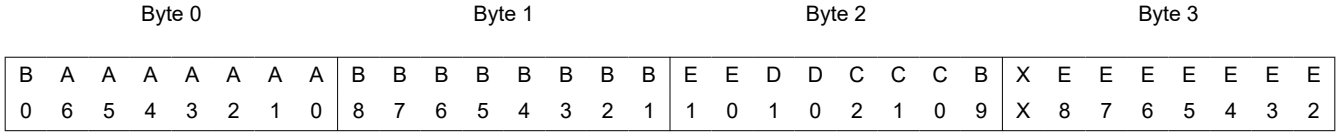
Big-endian memory



Little-endian register



Little-endian memory



LEGEND: X = not used, MS = most significant, LS = least significant

6.2.3 Character String Constants

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

- To initialize an array of characters. For example:

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

  When a string is used as an initializer, it is simply treated as an initialized array; each character is a separate initializer. For more information about initialization, see Section 6.10.
- 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! undefined behavior */
```

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.

The register conventions dictate how the compiler uses registers and how values are preserved across function calls. The following table shows the types of registers affected by these conventions. The "Register Usage" table summarizes how the compiler uses registers and whether their values are preserved across calls. For information about how values are preserved across calls, see Section 6.4.

Table 6-3. How Register Types Are Affected by the Conventions

Register Type	Description
Argument register	Passes arguments during a function call
Return register	Holds the return value from a function call
Expression register	Holds a value
Argument pointer	Used as a base value from which a function's parameters (incoming arguments) are accessed
Stack pointer	Holds the address of the top of the software stack
Link register	Contains the return address of a function call
Program counter	Contains the current address of code being executed

Table 6-4. Register Usage

Register	Alias	Usage	Preserved by Function[1]
R0	A1	Argument register, return register, expression register	Parent
R1	A2	Argument register, return register, expression register	Parent
R2	A3	Argument register, expression register	Parent
R3	A4	Argument register, expression register	Parent
R4	V1	Expression register	Child
R5	V2	Expression register	Child
R6	V3	Expression register	Child
R7	V4, AP	Expression register, argument pointer	Child
R8	V5	Expression register	Child
R9	V6	Expression register	Child
R10	V7	Expression register	Child
R11	V8	Expression register	Child
R12	V9, 1P	Expression register, instruction pointer	Parent
R13	SP	Stack pointer	Child[2]
R14	LR	Link register, expression register	Child
R15	PC	Program counter	N/A
CPSR		Current program status register	Child
SPSR		Saved program status register	Child

Table 6-5. VFP Register Usage

32-Bit Register	64-Bit Register	Usage	Preserved by Function[1]
FPSCR		Status register	N/A
S0	D0	Floating-point expression, return values, pass arguments	N/A
S1			
S2	D1	Floating-point expression, return values, pass arguments	N/A
S3			
S4	D2	Floating-point expression, return values, pass arguments	N/A
S5			
S6	D3	Floating-point expression, return values, pass arguments	N/A
S7			
S8	D4	Floating-point expression, pass arguments	N/A
S9			
S10	D5	Floating-point expression, pass arguments	N/A
S11			
S12	D6	Floating-point expression, pass arguments	N/A
S13			
S14	D7	Floating-point expression, pass arguments	N/A
S15			
S16	D8	Floating-point expression	Child
S17			
S18	D9	Floating-point expression	Child
S19			
S20	D10	Floating-point expression	Child
S21			
S22	D11	Floating-point expression	Child
S23			
S24	D12	Floating-point expression	Child
S25			
S26	D13	Floating-point expression	Child
S27			
S28	D14	Floating-point expression	Child
S29			
S30	D15	Floating-point expression	Child
S31			
	D16-D31	Floating-point expression	

Table 6-6. Neon Register Usage

64-Bit Register	Quad Register	Usage	Preserved by Function[1]
D0	Q0	SIMD register	N/A
D1			
D2	Q1	SIMD register	N/A
D3			
D4	Q2	SIMD register	N/A
D5			
D6	Q3	SIMD register	N/A
D7			
D8	Q4	SIMD register	Child
D9			
D10	Q5	SIMD register	Child
D11			
D12	Q6	SIMD register	Child
D13			
D14	Q7	SIMD register	Child
D15			
D16	Q8	SIMD register	N/A
D17			
D18	Q9	SIMD register	N/A
D19			
D20	Q10	SIMD register	N/A
D21			
D22	Q11	SIMD register	N/A
D23			
D24	Q12	SIMD register	N/A
D25			
D26	Q13	SIMD register	N/A
D27			
D28	Q14	SIMD register	N/A
D29			
D30	Q15	SIMD register	N/A
D31			
FPSCR		Status register	N/A

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.

The following sections use this terminology to describe the function-calling conventions of the C/C++ compiler:

- **Argument block**. The part of the local frame used to pass arguments to other functions. Arguments are passed to a function by moving them into the argument block rather than pushing them on the stack. The local frame and argument block are allocated at the same time.
- **Register save area**. The part of the local frame that is used to save the registers when the program calls the function and restore them when the program exits the function.
- **Save-on-call registers**. Registers R0-R3 and R12 (alternate names are A1-A4 and V9). The called function does not preserve the values in these registers; therefore, the calling function must save them if their values need to be preserved.

- **Save-on-entry registers**. Registers R4-R11 and R14 (alternate names are V1-V8 and LR). It is the called function's responsibility to preserve values in these registers. If the called function modifies these registers, it saves them when it gains control and preserves them when it returns control to the calling function.

For details on the calling conventions in EABI mode or when using a VFP coprocessor, refer to the EABI documentation located in the ARM Information Center.

Figure 6-5 illustrates a typical function call. In this example, arguments are passed to the function, and the function uses local variables and calls another function. The first four arguments are passed to registers R0-R3. This example also shows allocation of a local frame and argument block for the called function. Functions that have no local variables and do not require an argument block do not allocate a local frame.

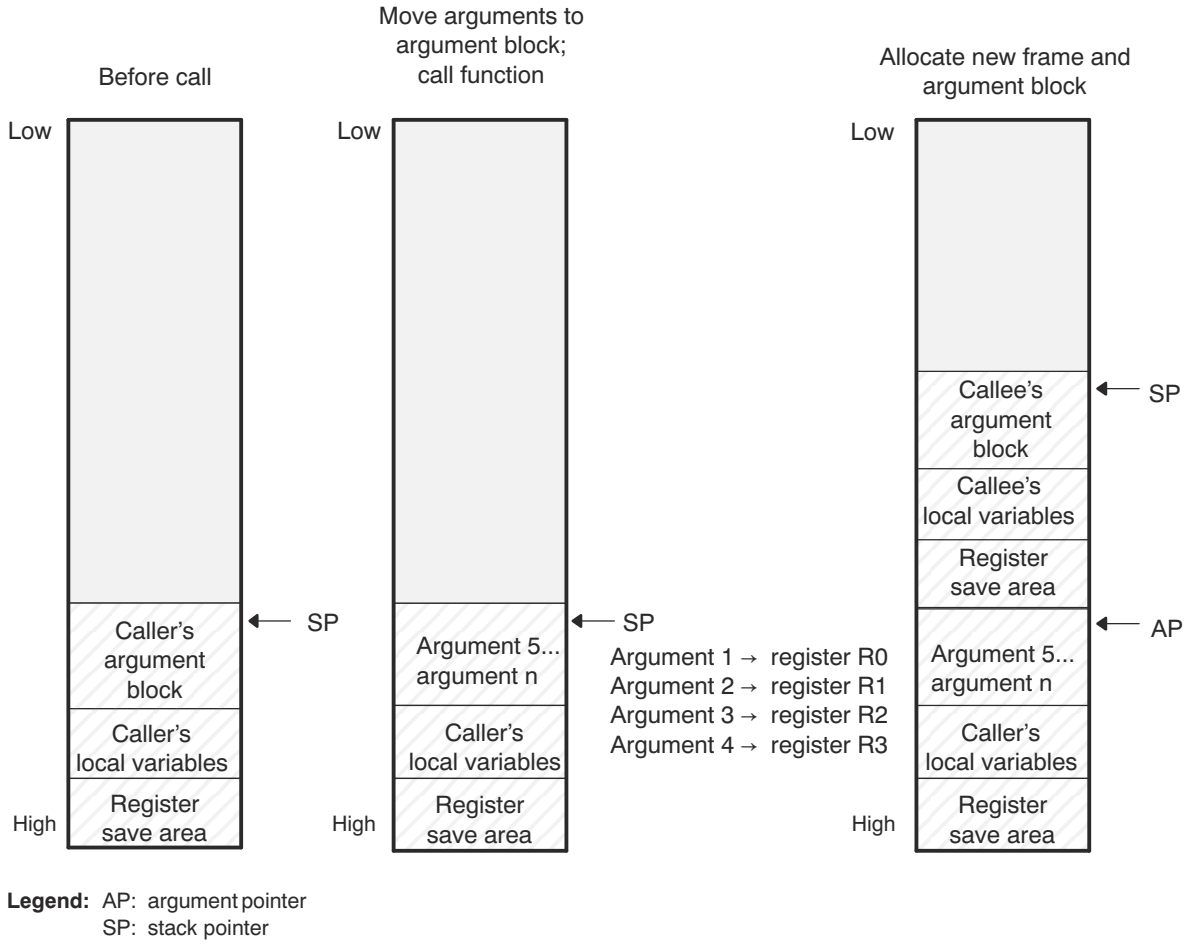


Figure 6-5. Use of the Stack During a Function Call

6.4.1 How a Function Makes a Call

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

1. The calling function (parent) is responsible for preserving any save-on-call registers across the call that are live across the call. (The save-on-call registers are R0-R3 and R12 (alternate names are A1-A4 and V9).)
2. If the called function (child) returns a structure, the caller allocates space for the structure and passes the address of that space to the called function as the first argument.
3. The caller places the first arguments in registers R0-R3, in that order. The caller moves the remaining arguments to the argument block in reverse order, placing the leftmost remaining argument at the lowest address. Thus, the leftmost remaining argument is placed at the top of the stack.
4. If arguments were stored onto the argument block in step 3, the caller reserves a word in the argument block for dual-state support. (See Section 6.11 for more information.)

6.4.2 How a Called Function Responds

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

1. If the function is declared with an ellipsis, it can be called with a variable number of arguments. The called function pushes these arguments on the stack if they meet both of these criteria:
   - The argument includes or follows the last explicitly declared argument.
   - The argument is passed in a register.
2. The called function pushes register values of all the registers that are modified by the function and that must be preserved upon exit of the function onto the stack. Normally, these registers are the save-on-entry registers (R4-R11 and R14 (alternate names are V1 to V8 and LR)) and the link register (R14) if the function contains calls. If the function is an interrupt, additional registers may need to be preserved. For more information, see Section 6.7.
3. The called function allocates memory for the local variables and argument block by subtracting a constant from the SP. This constant is computed with the following formula:

   size of all local variables + max = constant

   The *max* argument specifies the size of all parameters placed in the argument block for each call.
4. The called function executes the code for the function.
5. If the called function returns a value, it places the value in R0 (or R0 and R1 values).
6. If the called function returns a structure, it copies the structure to the memory block that the first argument, R0, points to. If the caller does not use the return value, R0 is set to 0. This directs the called function not to copy the return structure.

   In this way, the caller can be smart about telling the called function where to return the structure. For example, in the statement s = f(x), where s is a structure and f is a function that returns a structure, the caller can simply pass the address of s as the first argument and call f. The function f then copies the return structure directly into s, performing the assignment automatically.

   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 at the point where they are declared (so the function knows to copy the result).
7. The called function deallocates the frame and argument block by adding the constant computed in Step 3.
8. The called function restores all registers that were saved in Step 2.
9. The called function (_f) loads the program counter (PC) with the return address.

The following example is typical of how a called function responds to a call:

```
                                ; called function entry point
STMFD  SP!, {V1, V2, V3, LR} ; save V1, V2, V3, and LR
SUB    SP, SP, #16           ; allocate frame
...                          ; body of the function
ADD    SP, SP, #16           ; deallocate frame
LDMFD  SP!, {V1, V2, V3, PC} ; restore V1, V2, V3, and store LR
                                ; in the PC, causing a return
```

6.4.3 C Exception Handler Calling Convention

If a C exception handler calls other functions, the following must take place:

- The handler must set its own stack pointer.
- The handler saves all of the registers not preserved by the call: R0-R3, R-12, LR (R8-R12 saved by hardware for FIQ)
- Re-entrant exception handlers must save SPSR and LR.

6.4.4 Accessing Arguments and Local Variables

A function accesses its local nonregister variables indirectly through the stack pointer (SP or R13) and its stack arguments indirectly through the argument pointer (AP). If all stack arguments can be referenced with the SP, they are, and the AP is not reserved. The SP always points to the top of the stack (the most recently pushed value) and the AP points to the leftmost stack argument (the one closest to the top of the stack). For example:

```
LDR  A2 [SP, #4]  ; load local var from stack
LDR  A1 [AP, #0]  ; load argument from stack
```

Since the stack grows toward smaller addresses, the local and argument data on the stack for the C/C++ function is accessed with a positive offset from the SP or the AP register.

6.5 Accessing Linker Symbols in C and C++

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

6.6 Interfacing C and C++ With Assembly Language

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

- Use separate modules of assembled code and link them with compiled C/C++ modules (see Section 6.6.1).
- Use assembly language variables and constants in C/C++ source (see Section 6.6.3).
- Use inline assembly language embedded directly in the C/C++ source (see Section 6.6.5).
- Modify the assembly language code that the compiler produces (see Section 6.6.6).

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

Interfacing C/C++ with assembly language functions is straightforward if you follow the calling conventions defined in Section 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:

- You must preserve any dedicated registers modified by a function. Dedicated registers include:
  - Save-on-entry registers (R4-R11 (alternate names are V1 to V8 and LR))
  - Stack pointer (SP or R13)

  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.
- Interrupt routines must save *all* the registers they use. For more information, see Section 6.7.
- When you call a C/C++ function from assembly language, load the designated registers with arguments and push the remaining arguments onto the stack as described in Section 6.4.1.

  Remember that a function can alter any register not designated as being preserved without having to restore it. If the contents of any of these registers must be preserved across the call, you must explicitly save them.
- Functions must return values correctly according to their C/C++ declarations. Double values are returned in R0 and R1, and structures are returned as described in Step 2 of Section 6.4.1. Any other values are returned in R0.
- No assembly module should use the .cinit section for any purpose other than autoinitialization of global variables. The C/C++ startup routine assumes that the .cinit section consists *entirely* of initialization tables. Disrupting the tables by putting other information in .cinit can cause unpredictable results.
- The compiler assigns linknames to all external objects. Thus, when you write assembly language code, you must use the same linknames as those assigned by the compiler. See Section 5.15 for details.

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

  Likewise, to access a C/C++ function or object from assembly language, declare the C/C++ object with the .ref or .global directive in the assembly language module. This creates an undeclared external reference that the linker resolves.

6.6.2 Accessing Assembly Language Functions From C/C++

Functions defined in C++ that will be called from assembly should be defined as extern "C" in the C++ file. Functions defined in assembly that will be called from C++ must be prototyped as extern "C" in C++.

Example 6-1 illustrates a C++ function called main, which calls an assembly language function called asmfunc, Example 6-2. The asmfunc function takes its single argument, adds it to the C++ global variable called gvar, and returns the result.

Example 6-1. Calling an Assembly Language Function From a C/C++ Program

```
extern "C" {
extern int asmfunc(int a); /* declare external asm function */
int gvar = 0;              /* define global variable        */
}
void main()
{
   int I = 5;
   I = asmfunc(I);         /* call function normally       */
```

Example 6-2. Assembly Language Program Called by Example 6-1

```
          .global asmfunc
          .global gvar
asmfunc:
          LDR      r1, gvar_a
          LDR      r2, [r1, #0]
          ADD      r0, r0, r2
          STR      r0, [r1, #0]
          MOV      pc, lr
gvar_a    .field   gvar, 32
```

In the C++ program in Example 6-1, the extern "C" declaration tells the compiler to use C naming conventions (that is, no name mangling). When the linker resolves the .global _asmfunc reference, the corresponding definition in the assembly file will match.

The parameter i is passed in R0, and the result is returned in R0. R1 holds the address of the global gvar. R2 holds the value of gvar before adding the value i to it.

6.6.3 Accessing Assembly Language Variables From C/C++

It is sometimes useful for a C/C++ program to access variables or constants defined in assembly language. There are several methods that you can use to accomplish this, depending on where and how the item is defined: a variable defined in the .bss section, a variable not defined in the .bss section, or a linker symbol.

6.6.3.1 Accessing Assembly Language Global Variables

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

1. Use the .bss or .usect directive to define the variable.
2. 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 6-4 and Example 6-3 show how you can access a variable defined in .bss.

Example 6-3. Assembly Language Variable Program

```
        .bss     var,4,4  ; Define the variable
        .global  var      ; Declare the variable as external
```

Example 6-4. C Program to Access Assembly Language From Example 6-3

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

6.6.3.2 Accessing Assembly Language Constants

You can define global constants in assembly language by using the .set directive in combination with either the .def or .global directive, or you can define them in a linker command file using a linker assignment statement. These constants are accessible from C/C++ only with the use of special operators.

For **variables** defined in C/C++ or assembly language, the symbol table contains the *address of the value* contained by the variable. When you access an assembly variable by name from C/C++, the compiler gets the value using the address in the symbol table.

For **assembly constants**, however, the symbol table contains the actual *value* of the constant. The compiler cannot tell which items in the symbol table are addresses and which are values. If you access an assembly (or linker) constant by name, the compiler tries to use the value in the symbol table as an address to fetch a value. To prevent this behavior, you must use the & (address of) operator to get the value (_symval). In other words, if x is an assembly language constant, its value in C/C++ is &x. See the section on "Using Linker Symbols in C/C++ Applications" in the *ARM Assembly Language Tools User's Guide* for more examples that use _symval.

For more about symbols and the symbol table, refer to the section on "Symbols" in the *ARM Assembly Language Tools User's Guide*.

You can use casts and #defines to ease the use of these symbols in your program, as in Example 6-5 and Example 6-6.

Example 6-5. Accessing an Assembly Language Constant From C

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

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

```
_table_size  .set10000     ; define the constant
             .global _table_size  ; make it global
```

Because you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example 6-5, int is used. You can reference linker-defined symbols in a similar manner.

6.6.4 Sharing C/C++ Header Files With Assembly Source

You can use the .cdecls assembler directive to share C headers containing declarations and prototypes between C and assembly code. Any legal C/C++ can be used in a .cdecls block and the C/C++ declarations will cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code. For more information, see the C/C++ header files chapter in the *ARM Assembly Language Tools User's Guide*.

6.6.5 Using Inline Assembly Language

Within a C/C++ program, you can use the asm statement to insert a single line of assembly language into the assembly language file created by the compiler. A series of asm statements places sequential lines of assembly language into the compiler output with no intervening code. For more information, see Section 5.10.

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.

6.6.6 Modifying Compiler Output

You can inspect and change the compiler's assembly language output by compiling the source and then editing the assembly output file before assembling it. The C/C++ interlist feature can help you inspect compiler output. See Section 2.12.

6.7 Interrupt Handling

As long as you follow the guidelines in this section, you can interrupt and return to C/C++ code without disrupting the C/C++ environment. When the C/C++ environment is initialized, the startup routine does not enable or disable interrupts. If the system is initialized by way of a hardware reset, interrupts are disabled. If your system uses interrupts, you must handle any required enabling or masking of interrupts. Such operations have no effect on the C/C++ environment and are easily incorporated with asm statements or calling an assembly language function.

6.7.1 Saving Registers During Interrupts

When C/C++ code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any functions called by the routine. With the exception of banked registers, register preservation must be explicitly handled by the interrupt routine.

All banked registers are automatically preserved by the hardware (except for interrupts that are reentrant. If you write interrupt routines that are reentrant, you must add code that preserves the interrupt's banked registers.) Each interrupt type has a set of banked registers. For information about the interrupt types, see Section 5.11.16.

6.7.2 Using C/C++ Interrupt Routines

When C/C++ code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any functions called by the routine. Register preservation must be explicitly handled by the interrupt routine.

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

If a C/C++ interrupt routine does not call any other functions, only those registers that the interrupt handler uses are saved and restored. However, if a C/C++ interrupt routine does call other functions, these functions can modify unknown registers that the interrupt handler does not use. For this reason, the routine saves all the save-on-call registers if any other functions are called. (This excludes banked registers.) Do not call interrupt handling functions directly.

Interrupts can be handled directly with C/C++ functions by using the INTERRUPT pragma or the __interrupt keyword. For information, see Section 5.11.16 and Section 5.7.2, respectively.

6.7.3 Using Assembly Language Interrupt Routines

You can handle interrupts with assembly language code as long as you follow the same register conventions the compiler does. Like all assembly functions, interrupt routines can use the stack, access global C/C++ variables, and call C/C++ functions normally. When calling C/C++ functions, be sure that any save-on-call registers are preserved before the call because the C/C++ function can modify any of these registers. You do not need to save save-on-entry registers because they are preserved by the called C/C++ function.

6.7.4 How to Map Interrupt Routines to Interrupt Vectors

Note

This section does not apply to Cortex-M devices.

To map Cortex-A interrupt routines to interrupt vectors you need to include a intvecs.asm file. This file will contain assembly language directives that can be used to set up the ARM's interrupt vectors with branches to your interrupt routines. Follow these steps to use this file:

1. Using Example 6-7 as a guide, create intvecs.asm and include your interrupt routines. For each routine:
   a. At the beginning of the file, add a .global directive that names the routine.
   b. Modify the appropriate .word directive to create a branch to the name of your routine.
2. Assemble and link intvecs.asm with your applications code and with the compiler's linker control file (lnk16.cmd or lnk32.cmd). The control file contains a SECTIONS directive that maps the .intvecs section into the memory locations 0x00-0x1F.

For example, on an ARM v4 target, if you have written a C interrupt routine for the IRQ interrupt called c_intIRQ and an assembly language routine for the FIQ interrupt called tim1_int, you should create intvecs.asm as in Example 6-7.

Example 6-7. Sample intvecs.asm File

```
        .if __TI_EABI_ASSEMBLER
        .asg c_intIRQ, C_INTIRQ
        .else
        .asg _c_intIRQ, C_INTIRQ
        .endif
.global _c_int00
.global C_INTIRQ
.global tim1_int
.sect ".intvecs"
B _c_int00 ; reset interrupt
.word 0 ; undefined instruction interrupt
.word 0 ; software interrupt
.word 0 ; abort (prefetch) interrupt
.word 0 ; abort (data) interrupt
.word 0 ; reserved
B C_INTIRQ ; IRQ interrupt
B tim1_int ; FIQ interrupt
```

6.7.5 Using Software Interrupts

A software interrupt (SWI) is a synchronous exception generated by the execution of a particular instruction. Applications use software interrupts to request services from a protected system, such as an operating system, which can perform the services only while in a supervisor mode. Some ARM documentation uses the term Supervisor Calls (SVC) instead of "software interrupt".

A C/C++ application can invoke a software interrupt by associating a software interrupt number with a function name through use of the SWI_ALIAS pragma and then calling the software interrupt as if it were a function. For information, see Section 5.11.29.

Since a call to the software interrupt function represents an invocation of the software interrupt, passing and returning data to and from a software interrupt is specified as normal function parameter passing with the following restriction:

All arguments passed to a software interrupt must reside in the four argument registers (R0-R3). No arguments can be passed by way of a software stack. Thus, only four arguments can be passed unless:

- Floating-point doubles are passed, in which case each double occupies two registers.
- Structures are returned, in which case the address of the returned structure occupies the first argument register.

For Cortex-M architectures, C SWI handlers cannot return values. Values may be returned by SWI handlers on other architectures.

The C/C++ compiler also treats the register usage of a called software interrupt the same as a called function. It assumes that all save-on-entry registers () are preserved by the software interrupt and that save-on-call registers (the remainder of the registers) can be altered by the software interrupt.

6.7.6 Other Interrupt Information

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:

- It is your responsibility to handle any special masking of interrupts.
- A C/C++ interrupt routine cannot be called directly from C/C++ code.
- In a system reset interrupt, such as c_int00, you cannot assume that the run-time environment is set up; therefore, you *cannot allocate local variables*, and you *cannot save any information on the run-time stack*.
- In assembly language, remember to precede the name of a C/C++ interrupt with the appropriate linkname. For example, refer to c_int00 as _c_int00.
- When an interrupt occurs, the state of the processor (ARM or Thumb mode) is dependent on the device you are using. The compiler allows interrupt handlers to be defined in ARM or Thumb-2 mode. You should ensure the interrupt handler uses the proper mode for the device.
- The FIQ, supervisor, abort, IRQ, and undefined modes have separate stacks that are not automatically set up by the C/C++ run-time environment. If you have interrupt routines in one of these modes, you must set up the software stack for that mode. However, ARM Cortex-M processors have two stacks, and the main stack (MSP), which is used by IRQ (the only interrupt type for Cortex-M), is automatically handled by the compiler.
- Interrupt routines are not reentrant. If an interrupt routine enables interrupts of its type, it must save a copy of the return address and SPSR (the saved program status register) before doing so.
- Because a software interrupt is synchronous, the register saving conventions discussed in Section 6.7.1 can be less restrictive as long as the system is designed for this. A software interrupt routine generated by the compiler, however, follows the conventions in Section 6.7.1.

6.8 Intrinsic Run-Time-Support Arithmetic and Conversion Routines

The intrinsic run-time-support library contains a number of assembly language routines that provide arithmetic and conversion capability for C/C++ operations that the 32-bit and 16-bit instruction sets do not provide. These routines include integer division, integer modulus, and floating-point operations.

There are two versions of each of the routines:

- A 16-bit version to be called only from the 16-BIS (bit instruction set) state
- A 32-bit version only to be called from the 32-BIS state

These routines do not follow the standard C/C++ calling conventions in that the naming and register conventions are not upheld. Refer to the ARM Information Center for information on the EABI naming conventions.

6.8.1 CPSR Register and Interrupt Intrinsics

The intrinsics in Table 6-7 enable you to get/set the CPSR register and to enable/disable interrupts. All but the _call_swi() function are only available when compiling in ARM mode. Additional intrinsices for ARM assembly instructions are provided in Section 5.14.

Table 6-7. CPSR and Interrupt C/C++ Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction	Description
void **_call_swi(**unsigned int *src* **);**	**SWI $** *src*	Call a software interrupt. The *src* must be an immediate.
unsigned int *dst* = **_disable_FIQ( ) ;**	Cortex-R4/A8: **MRS** *dst* **, FAULTMASK** **CPSID** *f*	Disable FIQ interrupts and return previous FAULTMASK or CPSR setting.
unsigned int *dst* = **_disable_interrupts( ) ;**	Cortex-M0: **MRS** *dst* **, PRIMASK** **CPSID** *i* Cortex-M3/M4/R4/A8: **MRS** *dst* **, FAULTMASK** **CPSID** *f*	Disable all interrupts and return previous PRIMASK or FAULTMASK setting. The assembly instructions are dependent on the architecture.
unsigned int *dst* = **_disable_IRQ( ) ;**	**MRS** *dst* **, PRIMASK** **CPSID** *i*	Disable IRQ interrupts and return previous PRIMASK setting.
unsigned int *dst* = **_enable_FIQ( ) ;**	Cortex-R4/A8: **MRS** *dst* **, FAULTMASK** **CPSIE** *f*	Enable FIQ interrupts and return previous FAULTMASK or CPSR setting.
unsigned int *dst* = **_enable_interrupts( ) ;**	Cortex-M0: **MRS** *dst* **, PRIMASK** **CPSIE** *i* Cortex-M3/M4/R4/A8: **MRS** *dst* **, FAULTMASK** **CPSIE** *f*	Enable all interrupts and return previous PRIMASK or FAULTMASK setting. The assembly instructions are dependent on the architecture.
unsigned int *dst* = **_enable_IRQ( ) ;**	**MRS** *dst* **, PRIMASK** **CPSIE** *i*	Enable IRQ interrupts and return previous PRIMASK setting.
unsigned int *dst* = **_get_CPSR( ) ;**	**MRS** *dst* **, CPSR**	Get the CPSR register.
void **_restore_interupts(**unsigned int *src* **);**	Cortex-M0: **MSR PRIMASK** *src* Cortex-M3/M4: **MSR FAULTMASK** *src* Cortex-R4: **MSR CPSR_cf** , *src*	Restores interrupts to state indicated by value returned from _disable_interrupts. The assembly instructions are dependent on the architecture.
void **_set_CPSR(**unsigned int *src* **);**	**MSR CPSR** , *src*	Set the CPSR register.
void **_set_CPSR_flg(**unsigned int *src* **);**	**MSR** *dst* **, CPSR**	Set the CPSR flag bits. The *src* is rotated by the intrinsic.
unsigned int *dst* = **_set_interrupt_priority(** unsigned int *src* **) ;**	Cortex-M0/M3/M4 only: **MRS** *dst* **, BASEPRI** **MSR BASEPRI,** *src*	Set interrupt priority and return the previous setting.

6.9 Built-In Functions

Built-in functions are predefined by the compiler. They can be called like a regular function, but they do not require a prototype or definition. The compiler supplies the proper prototype and definition.

The ARM compiler supports the following built-in functions:

- The *__curpc* function, which returns the value of the program counter where it is called. The syntax of the function is:

```
void *__curpc(void);
```

- The *__run_address_check* function, which returns TRUE if the code performing the call is located at its run-time address, as assigned by the linker. Otherwise, FALSE is returned. The syntax of the function is:

```
int __run_address_check(void);
```

6.10 System Initialization

Before you can run a C/C++ program, you must create the C/C++ run-time environment. The C/C++ boot routine performs this task using a function called c_int00 (or _c_int00). The run-time-support source library, rts.src, contains the source to this routine in a module named boot.c (or boot.asm).

To begin running the system, the c_int00 function can be called by reset hardware. You must link the c_int00 function with the other object files. This occurs automatically when you use the --rom_model or --ram_model link option and include a standard run-time-support library as one of the linker input files.

When C/C++ programs are linked, the linker sets the entry point value in the executable output file to the symbol c_int00.

The c_int00 function performs the following tasks to initialize the environment:

1. Switches to the appropriate mode, reserves space for the run-time stack, and sets up the initial value of the stack pointer (SP). The stack is aligned on a 64-bit boundary.
2. Calls the function __TI_auto_init to perform the C/C++ autoinitialization.

   The __TI_auto_init function does the following tasks:
   - Processes the binit copy table, if present.
   - Performs C autoinitialization of global/static variables. For more information, see Section 6.10.3.
   - Calls C++ initialization routines for file scope construction from the global constructor table. For more information, see Section 6.10.3.6.
3. Calls the main() function to run the C/C++ program.

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the operations listed above to correctly initialize the C/C++ environment.

6.10.1 Boot Hook Functions for System Pre-Initialization

Boot hooks are points at which you may insert application functions into the C/C++ boot process. Default boot hook functions are provided with the run-time support (RTS) library. However, you can implement customized versions of these boot hook functions, which override the default boot hook functions in the RTS library if they are linked before the run-time library. Such functions can perform any application-specific initialization before continuing with the C/C++ environment setup.

Note that the TI-RTOS operating system uses custom versions of the boot hook functions for system setup, so you should be careful about overriding these functions if you are using TI-RTOS.

The following boot hook functions are available:

__mpu_init(): This function provides an interface for initializing the MPU, if MPU support is included. The __mpu_init() function is called after the stack pointer is initialized but before any C/C++ environment setup is performed. This function should not return a value.

_system_pre_init(): This function provides a place to perform application-specific initialization. It is invoked after the stack pointer is initialized but before any C/C++ environment setup is performed. For targets that include MPU support, this function is called after __mpu_init(). By default, _system_pre_init() should return a non-zero value. The default C/C++ environment setup is bypassed if _system_pre_init() returns 0.

_system_post_cinit(): This function is invoked during C/C++ environment setup, after C/C++ global data is initialized but before any C++ constructors are called. This function should not return a value.

The _c_int00( ) initialization routine also provides a mechanism for an application to perform the setup (set I/O registers, enable/disable timers, etc.) before the C/C++ environment is initialized.

6.10.2 Run-Time Stack

The run-time stack is allocated in a single continuous block of memory and grows down from high addresses to lower addresses. The SP points to the top of the stack.

The code does not check to see if the run-time stack overflows. Stack overflow occurs when the stack grows beyond the limits of the memory space that was allocated for it. Be sure to allocate adequate memory for the stack.

The stack size can be changed at link time by using the --stack_size link option on the linker command line and specifying the stack size as a constant directly after the option.

The C/C++ boot routine shipped with the compiler sets up the user/thread mode run-time stack. If your program uses a run-time stack when it is in other operating modes, you must also allocate space and set up the run-time stack corresponding to those modes.

EABI requires that 64-bit data (type long long and long double) be aligned at 64-bits. This requires that the stack be aligned at a 64-bit boundary at function entry so that local 64-bit variables are allocated in the stack with correct alignment. The boot routine aligns the stack at a 64-bit boundary.

6.10.3 Automatic Initialization of Variables

Any global variables declared as preinitialized must have initial values assigned to them before a C/C++ program starts running. The process of retrieving these variables' data and initializing the variables with the data is called autoinitialization. Internally, the compiler and linker coordinate to produce compressed initialization tables. Your code should not access the initialization table.

6.10.3.1 Zero Initializing Variables

In ANSI C, global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ compiler supports preinitialization of uninitialized variables by default. This can be turned off by specifying the linker option --zero_init=off.

Zero initialization takes place only if the --rom_model linker option, which causes autoinitialization to occur, is used. If you use the --ram_model option for linking, the linker does not generate initialization records, and the loader must handle both data and zero initialization.

6.10.3.2 Direct Initialization

The compiler uses direct initialization to initialize global variables. For example, consider the following C code:

```
int i    = 23;
int a[5] = { 1, 2, 3, 4, 5 };
```

The compiler allocates the variables 'i' and 'a[] to .data section and the initial values are placed directly.

```
        .global i
        .data
        .align  4
i:
        .field          23,32                   ; i @ 0
        .global a
        .data
        .align  4
a:
        .field          1,32                    ; a[0] @ 0
        .field          2,32                    ; a[1] @ 32
        .field          3,32                    ; a[2] @ 64
        .field          4,32                    ; a[3] @ 96
        .field          5,32                    ; a[4] @ 128
```

Each compiled module that defines static or global variables contains these .data sections. The linker treats the .data section like any other initialized section and creates an output section. In the load-time initialization model, the sections are loaded into memory and used by the program. See Section 6.10.3.5.

In the run-time initialization model, the linker uses the data in these sections to create initialization data and an additional compressed initialization table. The boot routine processes the initialization table to copy data from load addresses to run addresses. See Section 6.10.3.3.

6.10.3.3 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the --rom_model option.

Using this method, the linker creates a compressed initialization table and initialization data from the direct initialized sections in the compiled module. The table and data are used by the C/C++ boot routine to initialize variables in RAM using the table and data in ROM.

Figure 6-6 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

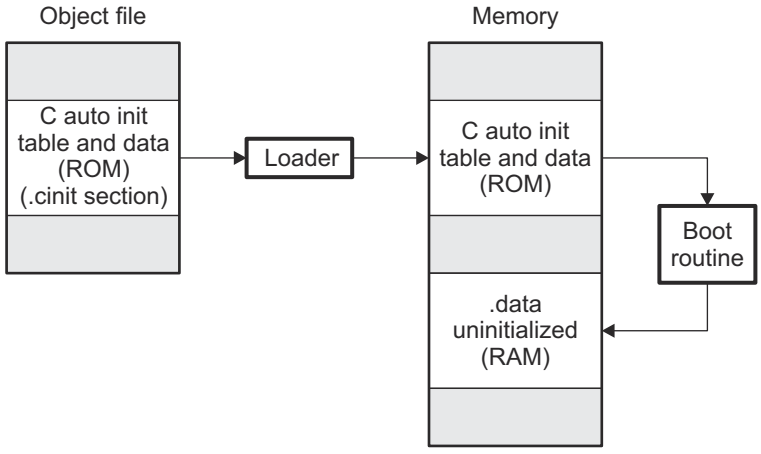


Figure 6-6. Autoinitialization at Run Time

6.10.3.4 Autoinitialization Tables

The compiled object files do not have initialization tables. The variables are initialized directly. The linker, when the --rom_model option is specified, creates C auto initialization table and the initialization data. The linker creates both the table and the initialization data in an output section named .cinit.

The autoinitialization table has the following format:

_TI_CINIT_Base:

32-bit load address	32-bit run address
⋮	⋮
32-bit load address	32-bit run address

_TI_CINIT_Limit:

The linker defined symbols __TI_CINIT_Base and __TI_CINIT_Limit point to the start and end of the table, respectively. Each entry in this table corresponds to one output section that needs to be initialized. The initialization data for each output section could be encoded using different encoding.

The load address in the C auto initialization record points to initialization data with the following format:

8-bit index	Encoded data

The first 8-bits of the initialization data is the handler index. It indexes into a handler table to get the address of a handler function that knows how to decode the following data.

The handler table is a list of 32-bit function pointers.

_TI_Handler_Table_Base:

32-bit handler 1 address
⋮
32-bit handler n address

_TI_Handler_Table_Limit:

The *encoded data* that follows the 8-bit index can be in one of the following format types. For clarity the 8-bit index is also depicted for each format.

6.10.3.4.1 Length Followed by Data Format

8-bit index	24-bit padding	32-bit length (N)	N byte initialization data (not compressed)

The compiler uses 24-bit padding to align the length field to a 32-bit boundary. The 32-bit length field encodes the length of the initialization data in bytes (N). N byte initialization data is not compressed and is copied to the run address as is.

The run-time support library has a function __TI_zero_init() to process this type of initialization data. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

6.10.3.4.2 Zero Initialization Format

8-bit index	24-bit padding	32-bit length (N)

The compiler uses 24-bit padding to align the length field to a 32-bit boundary. The 32-bit length field encodes the number of bytes to be zero initialized.

The run-time support library has a function __TI_zero_init() to process the zero initialization. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

6.10.3.4.3 Run Length Encoded (RLE) Format

8-bit index	Initialization data compressed using run length encoding

The data following the 8-bit index is compressed using Run Length Encoded (RLE) format. uses a simple run length encoding that can be decompressed using the following algorithm:

1. Read the first byte, Delimiter (D).
2. Read the next byte (B).
3. If B != D, copy B to the output buffer and go to step 2.
4. Read the next byte (L).
   a. If L == 0, then length is either a 16-bit, a 24-bit value, or we've reached the end of the data, read next byte (L).
      i. If L == 0, length is a 24-bit value or the end of the data is reached, read next byte (L).
         1. If L == 0, the end of the data is reached, go to step 7.
         2. Else L <<= 16, read next two bytes into lower 16 bits of L to complete 24-bit value for L.
      ii. Else L <<= 8, read next byte into lower 8 bits of L to complete 16-bit value for L.
   b. Else if L > 0 and L < 4, copy D to the output buffer L times. Go to step 2.
   c. Else, length is 8-bit value (L).
5. Read the next byte (C); C is the repeat character.
6. Write C to the output buffer L times; go to step 2.
7. End of processing.

The run-time support library has a routine __TI_decompress_rle24() to decompress data compressed using RLE. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

Note

RLE Decompression Routine

The previous decompression routine, __TI_decompress_rle(), is included in the run-time-support library for decompressing RLE encodings generated by older versions of the linker.

6.10.3.4.4 Lempel-Ziv-Storer-Szymanski Compression (LZSS) Format

8-bit index	Initialization data compressed using LZSS

The data following the 8-bit index is compressed using LZSS compression. The run-time support library has the routine __TI_decompress_lzss() to decompress the data compressed using LZSS. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

6.10.3.4.5 Sample C Code to Process the C Autoinitialization Table

The run-time support boot routine has code to process the C autoinitialization table. The following C code illustrates how the autoinitialization table can be processed on the target.

```
typedef void (*handler_fptr)(const unsigned char *in,
unsigned char *out);
#define HANDLER_TABLE   __TI_Handler_Table_Base
#pragma WEAK(HANDLER_TABLE)
extern  unsigned int   HANDLER_TABLE;
extern  unsigned char *__TI_CINIT_Base;
extern  unsigned char *__TI_CINIT_Limit;
void auto_initialize()
{
   unsigned char **table_ptr;
   unsigned char **table_limit;
   /*-------------------------------------------------------------*/
   /* Check if Handler table has entries.                         */
   /*-------------------------------------------------------------*/
   if (&__TI_Handler_Table_Base >= &__TI_Handler_Table_Limit)
      return;
   /*-------------------------------------------------------------*/
   /* Get the Start and End of the CINIT Table.                   */
   /*-------------------------------------------------------------*/
   table_ptr   = (unsigned char **)&__TI_CINIT_Base;
   table_limit = (unsigned char **)&__TI_CINIT_Limit;
   while (table_ptr < table_limit)
   {
      /*----------------------------------------------------------*/
      /* 1. Get the Load and Run address.                         */
      /* 2. Read the 8-bit index from the load address.           */
      /* 3. Get the handler function pointer using the index from */
      /*    handler table.                                        */
      /*----------------------------------------------------------*/
      unsigned char *load_addr   = *table_ptr++;
      unsigned char *run_addr    = *table_ptr++;
      unsigned char  handler_idx = *load_addr++;
      handler_fptr   handler     =
                  (handler_fptr)(&HANDLER_TABLE)[handler_idx];
      /*----------------------------------------------------------*/
      /* 4. Call the handler and pass the pointer to the load data*/
      /*    after index and the run address.                      */
      /*----------------------------------------------------------*/
      (*handler)((const unsigned char *)load_addr, run_addr);
   }
}
```

6.10.3.5 Initialization of Variables at Load Time

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the --ram_model option.

When you use the --ram_model link option, the linker does not generate C autoinitialization tables and data. The direct initialized sections (.data) in the compiled object files are combined according to the linker command file to generate initialized output sections. The loader loads the initialized output sections into memory. After the load, the variables are assigned their initial values.

Since the linker does not generate the C autoinitialization tables, no boot time initialization is performed.

Figure 6-7 illustrates the initialization of variables at load time.

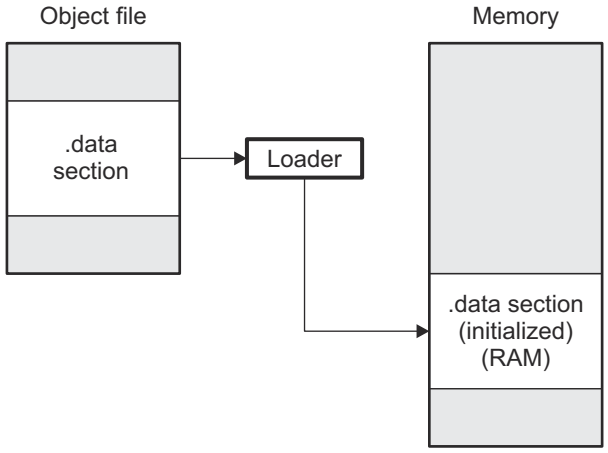


Figure 6-7. Initialization at Load Time

6.10.3.6 Global Constructors

All global C++ variables that have constructors must have their constructor called before main(). The compiler builds a table of global constructor addresses that must be called, in order, before main() in a section called .init_array. The linker combines the .init_array section form each input file to form a single table in the .init_array section. The boot routine uses this table to execute the constructors. The linker defines two symbols to identify the combined .init_array table as shown below. This table is not null terminated by the linker.

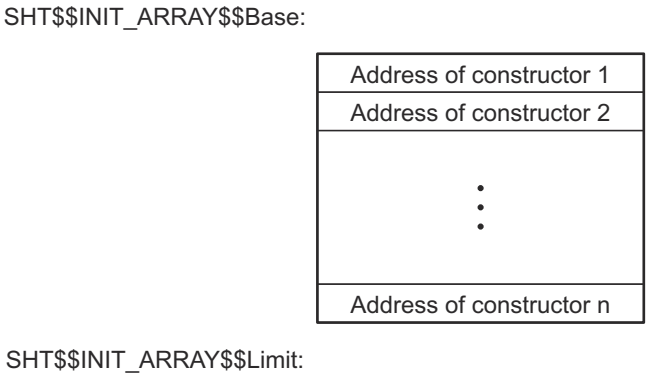


Figure 6-8. Constructor Table

6.10.4 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 6-9 shows the format of the .cinit section and the initialization records.

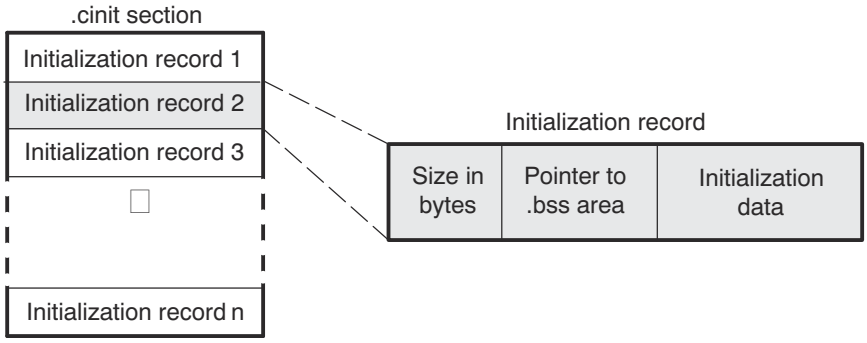


Figure 6-9. Format of Initialization Records in the .cinit Section

The fields of an initialization record contain the following information:

- The first field of an initialization record contains the size (in bytes) of the initialization data.The width of this field is one word (32-bit).
- The second field contains the starting address of the area within the .bss section where the initialization data must be copied.The width of this field is one word.
- The third field contains the data that is copied into the .bss section to initialize the variable.The width of this field is variable.

Each variable that must be autoinitialized has an initialization record in the .cinit section.

The following example shows initialized global variables defined in C.

```
int   i = 23;
int   a[5] = { 1, 2, 3, 4, 5 };
```

The corresponding initialization table is as follows. The section .cinit:c is a subsection in the .cinit section that contains all scalar data. The subsection is handled as one record during initialization, which minimizes the overall size of the .cinit section.

```
     .sect     ".cinit"          ; Initialization section
* Initialization record for variable i
     .align    4                 ; align on word boundary
     .field    4,32              ; length of data (1 word)
     .field    _i+0,32           ; address of i
     .field    23,32             ; _i @ 0
* Initialization record for variable a
     .sect     ".cinit"
     .align    4                 ; align on word boundary
     .field    IR1,32            ; Length of data (5 words)
     .field    _a+0,32           ; Address of a[ ]
     .field    1,32              ; _a[0] @ 0
     .field    2,32              ; _a[1] @ 32
     .field    3,32              ; _a[2] @ 64
     .field    4,32              ; _a[3] @ 96
     .field    5,32              ; _a[4] @ 128
IR1: .set      20                ; set length symbol
```

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 (as shown in the following figure). The constructors appear in the table after the .cinit initialization.

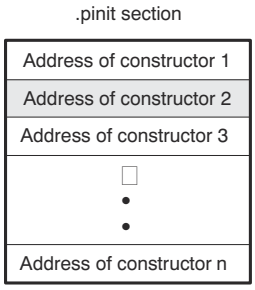


Figure 6-10. Format of Initialization Records in the .pinit Section

When you use the --rom_model or --ram_model option, the linker combines the .cinit sections from all the C/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 5.7.1.

6.11 Dual-State Interworking Under TIABI (Deprecated)

The ARM is a unique processor in that it gives you the performance of a 32-bit architecture with the code density of a 16-bit architecture. It supports a 16-bit instruction set and a 32-bit instruction set that allows switching dynamically between the two sets.

The instruction set that the ARM processor uses is determined by the state of the processor. The processor can be in 32-BIS (bit instruction set) state or 16-BIS state at any given time. The compiler allows you to specify whether a module should be compiled in 32- or 16-BIS state and allows functions compiled in one state to call functions compiled in the other state.

6.11.1 Level of Dual-State Support

By default, the compiler allows dual-state interworking between functions. However, the compiler allows you to alter the level of support to meet your particular needs.

In dual-state interworking, it is the called function's responsibility to handle the proper state changes required by the calling function. It is the calling function's responsibility to handle the proper state changes required to indirectly call a function (call it by address). Therefore, a function supports dual-state interworking if it provides the capability for functions requiring a state change to directly call the function (call it by name) and provides the mechanism to indirectly call functions involving state changes.

If a function does not support dual-state interworking, it cannot be called by functions requiring a state change and cannot indirectly call functions that support dual-state interworking. Regardless of whether a function supports dual-state interworking or not, it can directly or indirectly call certain functions:

- Directly call a function in the same state
- Directly call a function in a different state if that function supports dual-state interworking
- Indirectly call a function in the same state if that function does not support dual-state interworking

Given this definition of dual-state support, the ARM C/C++ compiler offers three levels of support. Use Table 6-8 to determine the best level of support to use for your code.

Table 6-8. Selecting a Level of Dual-State Support

If your code...	Use this level of support ...
Requires few state changes	Default
Requires many state changes	Optimized
Requires no state changes and has frequent indirect calls	None

Here is detailed information about each level of support:

- **Default**. Full dual-state interworking is supported. For each function that supports full dual-state interworking, the compiler generates code that allows functions requiring a state change to call the function, whether it is ever used or not. This code is placed in a different section from the section the actual function is in. If the linker determines that this code is never referenced, it does not link it into the final executable image. However, the mechanism used with indirect calls to support dual-state interworking is integrated into the function and cannot be removed by the linker, even if the linker determines that the mechanism is not needed.
- **Optimized**. Optimized dual-state interworking provides no additional functionality over the default level but optimizes the dual-state support code (in terms of code size and execution speed) for the case where a state change is required. It does this optimization by integrating the support into the function. Use the optimized level of support only when you know that a majority of the calls to this function require a state change. Even if the dual-state support code is never used, the linker cannot remove the code because it is integrated into the function. To specify this level of support, use the DUAL_STATE pragma. See Section 5.11.9 for more information.

- **None**. Dual-state interworking is disabled. This level is invoked with the -md shell option. Functions with this support can directly call the following functions:
  - Functions compiled in the same state
  - Functions in a different state that support dual-state interworking

  Functions with this support level can indirectly call only functions that do not require a state change and do not support dual-state interworking. Because functions with this support level do not provide dual-state interworking, they cannot be called by a function requiring a state change.

  Use this support level if you do not require dual-state interworking, have frequent indirect calls, and cannot tolerate the additional code size or speed incurred by the indirect calls supporting dual-state interworking.

When a program does not require any state changes, the only difference between specifying no support and default support is that indirect calls are more complex in the default support level.

6.11.2 Implementation

Dual-state support is implemented by providing an alternate entry point for a function. This alternate entry point is used by functions requiring a state change. Dual-state support handles the change to the correct state and, if needed, changes the function back to the state of the caller when it returns. Also, indirect calls set up the return address so that once the called function returns, the state can be reliably changed back to that of the caller.

6.11.2.1 Naming Conventions for Entry Points

The ARM compiler reserves the name space of all identifiers beginning with an underscore (_) or a dollar sign ($). In this dual-state support scheme, all 32-BIS state entry points begin with an underscore, and all 16-BIS state entry points begin with a dollar sign. All other compiler-generated identifiers, which are independent of the state of the processor, begin with an underscore. By this convention, all direct calls within a 16-bit function refer to the entry point beginning with a dollar sign and all direct calls within a 32-bit function refer to the entry point beginning with an underscore.

6.11.2.2 Indirect Calls

Addresses of functions taken in 16-BIS state use the address of the 16-BIS state entry point to the function (with bit 0 of the address set). Likewise, addresses of functions taken in 32-BIS state use the address of the 32-BIS state entry point (with bit 0 of the address cleared). Then all indirect calls are performed by loading the address of the called function into a register and executing the branch and exchange (BX) instruction. This automatically changes the state and ensures that the code works correctly, regardless of what state the address was in when it was taken.

The return address must also be set up so that the state of the processor is consistent and known upon return. Bit 0 of the address is tested to determine if the BX instruction invokes a state change. If it does not invoke a state change, the return address is set up for the state of the function. If it does invoke a change, the return address is set up for the alternate state and code is executed to return to the function's state.

Because the entry point into a function depends upon the state of the function that takes the address, it is more efficient to take the address of a function when in the same state as that function. This ensures that the address of the actual function is used, not its alternate entry point. Because the indirect call can invoke a state change itself, entering a function through its alternate entry point, even if calling it from a different state, is unnecessary.

Example 6-8 shows sum( ) calling max( ) with code that is compiled for the 16-BIS state and supports dual-state interworking. The sum( ) function is compiled with the -code_state=16 option, which creates 16-bit instructions for pre-UAL assembly code. (Refer to the *ARM Assembly Language Tools User's Guide* for information on UAL syntax.) Example 6-11 shows the same function call with code that is compiled for the 32-BIS state and supports dual-state interworking. Function max( ) is compiled without the -code_state=16 option, creating 32-bit instructions.

Example 6-8. C Code Compiled for 16-BIS State: sum( )

```
int total = 0;
sum(int val1, int val2)
{
   int val = max(val1, val2);
   total += val;
}
```

Example 6-9. 16-Bit Assembly Program for Example 6-8

```
;***************************************************************
;* FUNCTION VENEER: _sum                                       *
;***************************************************************
_sum:
      .state32
      STMFD  sp!, {lr}
      ADD    lr, pc, #1
      BX     lr
      .state16
      BL     $sum
      BX     pc
      NOP
      .state32
      LDMFD  sp!, {pc}
      .state16
      .sect   ".text"
      .global sum
;***************************************************************
;* FUNCTION DEF: $sum                                          *
;***************************************************************
$sum:
      PUSH   {LR}
      BL     $max
      LDR    A2, CON1 ; {_total+0}
      LDR    A3, [A2, #0]
      ADD    A1, A1, A3
      STR    A1, [A2, #0]
      POP    {PC}
;***************************************************************
;* CONSTANT TABLE                                              *
;***************************************************************
      .sect  ".text"
      .align 4
CON1: .field _total, 32
```

Example 6-10. C Code Compiled for 32-BIS State: sum( )

```
int max(int a, int b)
{
   return a < b ? b : a;
}
```

Example 6-11. 32-Bit Assembly Program for Example 6-10

```
;*****************************************************
;* FUNCTION VENEER: $max                             *
;*****************************************************
$max:
      .state16
      BX     pc
      NOP
      .state32
      B      _max
      .text
      .global _max
;*****************************************************
;* FUNCTION DEF: _max                                *
;*****************************************************
_max:
      CMP    A1, A2
      MOVLE  A1, A2
      BX     LR
```

Since sum( ) is a 16-bit function, its entry point is $sum. Because it was compiled for dual-state interworking, an alternate entry point, _sum, located in a different section is included. All calls to sum( ) requiring a state change use the _sum entry point.

The call to max( ) in sum( ) references $max, because sum( ) is a 16-bit function. If max( ) were a 16-bit function, sum( ) would call the actual entry point for max( ). However, since max( ) is a 32-bit function, $max is the alternate entry point for max( ) and handles the state change required by sum( ).

Chapter 7

Using Run-Time-Support Functions and Building Libraries

Some of the features of C/C++ (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are provided as an ANSI/ISO C/C++ standard library, rather than as part of the compiler itself. The TI implementation of this library is the run-time-support library (RTS). The C/C++ compiler implements the ISO standard library except for those facilities that handle exception conditions, signal and locale issues (properties that depend on local language, nationality, or culture). Using the ANSI/ISO standard library ensures a consistent set of functions that provide for greater portability.

In addition to the ANSI/ISO-specified functions, the run-time-support library includes routines that give you processor-specific commands and direct C language I/O requests. These are detailed in Section 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.

7.1 C and C++ Run-Time Support Libraries 174
7.2 The C I/O Functions 178
7.3 Handling Reentrancy (_register_lock() and _register_unlock() Functions) 190
7.4 Library-Build Process 191

7.1 C and C++ Run-Time Support Libraries

ARM compiler releases include pre-built run-time support (RTS) libraries that provide all the standard capabilities. Separate libraries are provided for each mode, big and little endian support, each ABI (compiler version 4.1.0 and later), various architectures, and C++ exception support. See Section 7.1.9 for information on the library-naming conventions.

The run-time-support library contains the following:

- ANSI/ISO C/C++ standard library
- C I/O library
- Low-level support functions that provide I/O to the host operating system
- Fundamental arithmetic routines
- System startup routine, _c_int00
- Compiler helper functions (to support language features that are not directly efficiently expressible in C/C++)

The run-time-support libraries do not contain functions involving signals and locale issues.

The C++ library supports wide chars, in that template functions and classes that are defined for char are also available for wide char. For example, wide char stream classes wios, wiostream, wstreambuf and so on (corresponding to char classes ios, iostream, streambuf) are implemented. However, there is no low-level file I/O for wide chars. Also, the C library interface to wide char support (through the C++ headers <cwchar> and <cwctype>) is limited as described in Section 5.1.

TI does not provide documentation that covers the functionality of the C++ library. TI suggests referring to one of the following sources:

- *The Standard C++ Library: A Tutorial and Reference*, Nicolai M. Josuttis, Addison-Wesley, ISBN 0-201-37926-0
- *The C++ Programming Language* (Third or Special Editions), Bjarne Stroustrup, Addison-Wesley, ISBN 0-201-88954-4 or 0-201-70073-5

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*ARM 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.

If you want to link object files created with the TI CodeGen tools with object files generated by other compiler tool chains, the ARM standard specifies that you should define the _AEABI_PORTABILITY_LEVEL preprocessor symbol as follows before #including any standard header files, such as <stdlib.h>.

```
#define _AEABI_PORTABILITY_LEVEL 1
```

This definition enables full portability. Defining the symbol to 0 specifies that the "C standard" portability level will be used.

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 TI_ARM_C_DIR environment variable to the include directory where the tools are installed.

The following header files provide TI extensions to the C standard:

- cpy_tbl.h -- Declares the copy_in() RTS function, which is used to move code or data from a load location to a separate run location at run-time. This function helps manage overlays.
- file.h -- Declares functions used by low-level I/O functions in the RTS library.
- _lock.h -- Used when declaring system-wide mutex locks. This header file is deprecated; use _reg_mutex_api.h and _mutex.h instead.
- memory.h -- Provides the memalign() function, which is not required by the C standard.
- _mutex.h -- Declares functions used by the RTS library to help facilitate mutexes for specific resources that are owned by the RTS. For example, these functions are used for heap or file table allocation.
- _pthread.h -- Declares low-level mutex infrastructure functions and provides support for recursive mutexes.
- _reg_mutex_api.h -- Declares a function that can be used by an RTOS to register an underlying lock mechanism and/or thread ID mechanism that is implemented in the RTOS but is called indirectly by the RTS' _mutex.h functions.
- _reg_synch_api.h -- Declares a function that can be used by an RTOS to register an underlying cache synchronization mechanism that is implemented in the RTOS but is called indirectly by the RTS' _data_synch.h functions.
- strings.h -- Provides additional string functions, including bcmp(), bcopy(), bzero(), ffs(), index(), rindex(), strcasecmp(), and strncasecmp().

7.1.3 Modifying a Library Function

You can inspect or modify library functions by examining the source code in the lib/src subdirectory of the compiler installation. For example, `C:\ti\ccsv7\tools\compiler\arm_#.#.#\lib\src`.

One you have located the relevant source code, change the specific function file and rebuild the library.

You can use this source tree to rebuild the rtsv4_A_be_eabi.lib library or to build a new library. See Section 7.1.9 for details on library naming and Section 7.4 for details on building

7.1.4 Support for String Handling

The RTS library provides the standard C header file <string.h> as well as the POSIX header file <strings.h>, which provides additional functions not required by the C standard. The POSIX header file <strings.h> provides:

- bcmp(), which is equivalent to memcmp()
- bcopy(), which is equivalent to memmove()
- bzero(), which is equivalent to memset(.., 0, ...);
- ffs(), which finds the first bit set and returns the index of that bit
- index(), which is equivalent to strchr()
- rindex(), which is equivalent to strrchr()
- strcasecmp() and strncasecmp(), which perform case-insensitive string comparisons

In addition, the header file <string.h> provides one additional function not required by the C standard.

- strdup(), which duplicates a string by dynamically allocating memory (as if by using malloc) and copying the string to this allocated memory

7.1.5 Minimal Support for Internationalization

The library includes the header files <locale.h>, <wchar.h>, and <wctype.h>, which provide APIs to support non-ASCII character sets and conventions. Our implementation of these APIs is limited in the following ways:

- The library has minimal support for wide and multibyte characters. The type wchar_t is implemented as int. The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h> but does not include all the functions specified in the standard. See Section 5.6 for more information about extended character sets.
- The C library includes the header file <locale.h> but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hard-coded to the behavior of the C locale, and attempting to install a different locale via a call to setlocale() will return NULL.

7.1.6 Support for Time and Clock Functions

The compiler RTS library supports two low-level time-related standard C functions in `time.h`:

- `clock_t clock(void);`
- `time_t time(time_t *timer);`

The time() function returns the wall-clock time. The clock() function returns the number of clock cycles since the program began executing; it has nothing to do with wall-clock time.

The default implementations of these functions require that the program be run under CCS or a similar tool that supports the CIO System Call Protocol. If CIO is not available and you need to use one of these functions, you must provide your own definition of the function.

The clock() function returns the number of clock cycles since the program began executing. This information might be available in a register on the device itself, but the location varies from platform to platform. The compiler's RTS library provides an implementation that uses the CIO System Call Protocol to communicate with CCS, which figures out how to compute the right value for this device.

If CCS is not available, you must provide an implementation of the clock() function that gathers clock cycle information from the appropriate location on the device.

The time() function returns the real-world time, in terms of seconds since an epoch.

On many embedded systems, there is no internal real-world clock, so a program needs to discover the time from an external source. The compiler's RTS library provides an implementation that uses the CIO System Call Protocol to communicate with CCS, which provides the real-world time.

If CCS is not available, you must provide an implementation of the time() function that finds the time from some other source. If the program is running under an operating system, that operating system should provide an implementation of time().

The time() function returns the number of seconds since an *epoch*. On POSIX systems, the epoch is defined as the number of seconds since midnight UTC January 1, 1970. However, the C standard does not require any particular epoch, and the default TI version of time() uses a different epoch: midnight UTC-6 (CST) Jan 1, 1900. Also, the default TI time_t type is a 32-bit type, while POSIX systems typically use a 64-bit time_t type.

The RTS library provides a non-default implementation of the time() function that uses the midnight UTC January 1, 1970 epoch and the 64-bit time_t type, which is then a typedef for __type64_t.

If your code works with raw time values, you can handle the epoch issue in one of the following ways:

- Use the default time() function with the 1900 epoch and 32-bit time_t type. A separate __time64_t type is available in this case.
- Define the macro __TI_TIME_USES_64. The time() function will use the 1970 epoch and the 64-bit time_t type, in which case time_t is a typedef for __type64_t.

Table 7-1. Differences between __time32_t and __time64_t

	__time32_t	__time64_t
Epoch (start)	Jan. 1, 1900 CST-0600	Jan. 1, 1970 UTC-0000
End date	Feb. 7, 2036 06:28:14	year 292277026596
Sign	Unsigned, so cannot represent dates before the epoch.	Signed, so can represent dates before the epoch.

7.1.7 Allowable Number of Open Files

In the <stdio.h> header file, the value for the macro FOPEN_MAX has the value of the macro _NFILE, which is set to 10. The impact is that you can only have 10 files simultaneously open at one time (including the pre-defined streams - stdin, stdout, stderr).

The C standard requires that the minimum value for the FOPEN_MAX macro is 8. The macro determines the maximum number of files that can be opened at one time. The macro is defined in the stdio.h header file and can be modified by changing the value of the _NFILE macro and recompiling the library.

7.1.8 Nonstandard Header Files in the Source Tree

The source code in the lib/src subdirectory of the compiler installation contains these non-ANSI include files that are used to build the library:

- The *values.h* file contains the definitions necessary for recompiling the trigonometric and transcendental math functions. If necessary, you can customize the functions in values.h.
- The *file.h* file includes macros and definitions used for low-level I/O functions.
- The *format.h* file includes structures and macros used in printf and scanf.
- The *470cio.h* file includes low-level, target-specific C I/O macro definitions. If necessary, you can customize 470cio.h.
- The *rtti.h* file includes internal function prototypes necessary to implement run-time type identification.
- The *vtbl.h* file contains the definition of a class's virtual function table format.

7.1.9 Library Naming Conventions

By default, the linker uses automatic library selection to select the correct run-time-support library (see Section 4.3.1.1) for your application. If you select the library manually, you must select the matching library using a naming scheme like the following:

rtsArchVersion_mode_endian[_n][_v*n*]_*abi*[_eh].lib

ArchVersion	The version of the ARM architecture that the library was built for. This can be one of the following: v4, v5, v6, v6M0, v7A8, v7R4, v7R5, or v7M3.	
mode	Indicates compilation mode:	
	T	Thumb mode
	A	ARM mode
endian	Indicates endianness:	
	le	Little-endian library
	be	Big-endian library
n	Indicates the library contains NEON support.	
v*n*	Indicates the library has VFP support. *n* designates the version. Current values are:	
	2	VFPv2
	3	VFPv3
	3D16	VFPv3D16
abi	Indicates the application binary interface (ABI) used. Although the TI_ARM9_ABI and TIARM ABIs are no longer supported, the library filename still contains "_eabi" to distinguish the EABI libraries from older libraries.	
	eh	Indicates the library has exception handling support

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.

The formatting rules for long long data types require ll (lowercase LL) in the format string. For example:

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

Note

Debugger Required for Default HOST: For the default HOST device to work, there must be a debugger to handle the C I/O requests; the default HOST device cannot work by itself in an embedded system. To work in an embedded system, you will need to provide an appropriate driver for your system.

Note

C I/O Mysteriously Fails: If there is not enough space on the heap for a C I/O buffer, operations on the file will silently fail. If a call to printf() mysteriously fails, this may be the reason. The heap needs to be at least large enough to allocate a block of size BUFSIZ (defined in stdio.h) for every file on which I/O is performed, including stdout, stdin, and stderr, plus allocations performed by the user's code, plus allocation bookkeeping overhead. Alternately, declare a char array of size BUFSIZ and pass it to setvbuf to avoid dynamic allocation. To set the heap size, use the --heap_size option when linking (refer to the *Linker Description* chapter in the *ARM Assembly Language Tools User's Guide*).

Note

Open Mysteriously Fails: The run-time support limits the total number of open files to a small number relative to general-purpose processors. If you attempt to open more files than the maximum, you may find that the open will mysteriously fail. You can increase the number of open files by extracting the source code from rts.src and editing the constants controlling the size of some of the C I/O data structures. The macro _NFILE controls how many FILE (fopen) objects can be open at one time (stdin, stdout, and stderr count against this total). (See also FOPEN_MAX.) The macro _NSTREAM controls how many low-level file descriptors can be open at one time (the low-level files underlying stdin, stdout, and stderr count against this total). The macro _NDEVICE controls how many device drivers are installed at one time (the HOST device counts against this total).

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 for each module that references a function.
- Allow for 320 bytes of heap space for each I/O stream used in your program. A stream is a source or destination of data that is associated with a peripheral, such as a terminal or keyboard. Streams are buffered using dynamically allocated memory that is taken from the heap. More heap space may be required to support programs that use additional amounts of dynamically allocated memory (calls to malloc()). To set the heap size, use the --heap_size option when linking; see Table 2-22.

For example, given the following C program in a file named main.c:

```
#include <stdio.h>
void main()
{
   FILE *fid;
   fid = fopen("myfile","w");
   fprintf(fid,"Hello, world\n");
   fclose(fid);
   printf("Hello again, world\n");
}
```

Issuing the following compiler command compiles, links, and creates main.out from the run-time-support library:

```
armcl main.c --run_linker --heap_size=400 --library=rtsv4_A_be_eabi.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.

7.2.1.1 Formatting and the Format Conversion Buffer

The internal routine behind the C I/O functions—such as printf(), vsnprintf(), and snprintf()—reserves stack space for a format conversion buffer. The buffer size is set by the macro FORMAT_CONVERSION_BUFFER, which is defined in format.h. Consider the following issues before reducing the size of this buffer:

- The default buffer size is 510 bytes. If MINIMAL is defined, the size is set to 32, which allows integer values without width specifiers to be printed.
- Each conversion specified with %xxxx (except %s) must fit in FORMAT_CONVERSION_BUFSIZE. This means any individual formatted float or integer value, accounting for width and precision specifiers, needs to fit in the buffer. Since the actual value of any representable number should easily fit, the main concern is ensuring the width and/or precision size meets the constraints.
- The length of converted strings using %s are unaffected by any change in FORMAT_CONVERSION_BUFSIZE. For example, you can specify `printf("%s value is %d", some_really_long_string, intval)` without a problem.
- The constraint is for each individual item being converted. For example a format string of `%d item1 %f item2 %e item3` does not need to fit in the buffer. Instead, each converted item specified with a % format must fit.
- There is no buffer overrun check.

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 choose any name except for HOST.

DEV_open

Open File for I/O

Syntax

int DEV_open (const char * *path* **, unsigned** *flags* **, int** *llv_fd* **);**

Description

This function finds a file matching *path* and opens it for I/O as requested by *flags*.

- The *path* is the filename of the file to be opened. If the name of a file passed to open has a device prefix, the device prefix will be stripped by open, so DEV_open will not see it. (See Section 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. The file descriptor need not be unique across devices. The device file descriptor is used only by low-level functions when calling the device-driver-level functions. The low-level function `open` allocates its own unique file descriptor for the high-level functions to call the low-level functions. Code that uses only high-level I/O functions need not be aware of these file descriptors.

DEV_close

Close File for I/O

Syntax

int DEV_close (int *dev_fd* **);**

Description

This function closes a valid open file descriptor.

On some devices, DEV_close may need to be responsible for checking if this is the last file descriptor pointing to a file that was unlinked. If so, it is responsible for ensuring that the file is actually removed from the device and the resources reclaimed, if appropriate.

Return Value

This function should return -1 to indicate an error if the file descriptor is invalid in some way, such as being out of range or already closed, but this is not required. The user should not call close() with an invalid file descriptor.

DEV_read

Read Characters from a File

Syntax

int DEV_read (int *dev_fd* **, char *** *buf* **, unsigned** *count* **);**

Description

The read function reads *count* bytes from the input file associated with *dev_fd*.

- The *dev_fd* is the number assigned by open to an opened file.
- The *buf* is where the read characters are placed.
- The *count* is the number of characters to read from the file.

Return Value

This function must return -1 to indicate an error if for some reason no bytes could be read from the file. This could be because of an attempt to read from a O_WRONLY file, or for device-specific reasons.

If count is 0, no bytes are read and this function returns 0.

This function returns the number of bytes read, from 0 to count. 0 indicates that EOF was reached before any bytes were read. It is not an error to read less than count bytes; this is common if the are not enough bytes left in the file or the request was larger than an internal device buffer size.

DEV_write

Write Characters to a File

Syntax

int DEV_write (int *dev_fd* **, const char *** *buf* **, unsigned** *count* **);**

Description

This function writes *count* bytes to the output file.

- The *dev_fd* is the number assigned by open to an opened file.
- The *buffer* is where the write characters are placed.
- The *count* is the number of characters to write to the file.

Return Value

This function must return -1 to indicate an error if for some reason no bytes could be written to the file. This could be because of an attempt to read from a O_RDONLY file, or for device-specific reasons.

DEV_lseek

Set File Position Indicator

Syntax

off_t DEV_lseek (int *dev_fd* **, off_t** *offset* **, int** *origin* **);**

Description

This function sets the file's position indicator for this file descriptor as lseek.

If lseek is supported, it should not allow a seek to before the beginning of the file, but it should support seeking past the end of the file. Such seeks do not change the size of the file, but if it is followed by a write, the file size will increase.

Return Value

If successful, this function returns the new value of the file position indicator.

This function must return -1 to indicate an error if for some reason no bytes could be written to the file. For many devices, the lseek operation is nonsensical (e.g. a computer monitor).

DEV_unlink

Delete File

Syntax

int DEV_unlink (const char * *path* **);**

Description

Remove the association of the pathname with the file. This means that the file may no longer be opened using this name, but the file may not actually be immediately removed.

Depending on the device, the file may be immediately removed, but for a device which allows open file descriptors to point to unlinked files, the file will not actually be deleted until the last file descriptor is closed. See Section 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 does not exist, or the new name already exists.

Note

It is inadvisable to allow renaming a file so that it is on a different device. In general this would require a whole file copy, which may be more expensive than you expect.

If successful, this function returns 0.

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.

Note

Use Unique Function Names

The function names open, read, write, close, lseek, rename, and unlink are used by the low-level routines. Use other names for the device-level functions that you write.

Use the low-level function add_device() to add your device to the device_table. The device table is a statically defined array that supports *n* devices, where *n* is defined by the macro _NDEVICE found in stdio.h/cstdio.

The first entry in the device table is predefined to be the host device on which the debugger is running. The low-level routine add_device() finds the first empty position in the device table and initializes the device fields with the passed-in arguments. For a complete description, see the add_device function.

Example 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 */
    /* does not buffer stderr at all. This means that there will be one call */
    /* to write() for each character in the message.                      */
    /*--------------------------------------------------------------------*/
    if (setvbuf(stderr, NULL, _IONBF, 0))
    {
        puts("Failed to setvbuf stderr");
        exit(EXIT_FAILURE);
    }
    /*--------------------------------------------------------------------*/
    /* Try it out!                                                        */
    /*--------------------------------------------------------------------*/
    printf("This goes to stdout\n");
    fprintf(stderr, "This goes to stderr\n"); }
```

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 the lib/src subdirectory of the compiler installation)

Description

The add_device function adds a device record to the device table allowing that device to be used for I/O from C. The first entry in the device table is predefined to be the HOST device on which the debugger is running. The function add_device() finds the first empty position in the device table and initializes the fields of the structure that represent a device.

To open a stream on a newly added device use fopen( ) with a string of the format *devicename* **:** *filename* as the first argument.

- The *name* is a character string denoting the device name. The name is limited to 8 characters.
- The *flags* are device characteristics. The flags are as follows:

  _SSA Denotes that the device supports only one open stream at a time

  _MSA Denotes that the device supports multiple open streams

  More flags can be added by defining them in file.h.
- The *dopen*, *dclose*, *dread*, *dwrite*, *dlseek*, *dunlink*, and *drename* specifiers are function pointers to the functions in the device driver that are called by the low-level functions to perform I/O on the specified device. You must declare these functions with the interface specified in Section 7.2.2. The device driver for the HOST that the ARM debugger is run on are included in the C I/O library.

Return Value

The function returns one of the following values:

0	if successful
-1	on failure

add_device (continued)

Add Device to Device Table

Example

Example 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, SYS/BIOS applications have multiple threads which need to modify the same global program state, such as the CIO buffer, so reentrancy is a concern.

Part of the problem of reentrancy remains your responsibility, but the run-time-support environment does provide rudimentary support for multi-threaded reentrancy by providing support for critical sections. This implementation does not protect you from reentrancy issues such as calling run-time-support functions from inside interrupts; this remains your responsibility.

The run-time-support environment provides hooks to install critical section primitives. By default, a single-threaded model is assumed, and the critical section primitives are not employed. In a multi-threaded system such as SYS/BIOS, the kernel arranges to install semaphore lock primitive functions in these hooks, which are then called when the run-time-support enters code that needs to be protected by a critical section.

Throughout the run-time-support environment where a global state is accessed, and thus needs to be protected with a critical section, there are calls to the function _lock(). This calls the provided primitive, if installed, and acquires the semaphore before proceeding. Once the critical section is finished, _unlock() is called to release the semaphore.

Usually SYS/BIOS is responsible for creating and installing the primitives, so you do not need to take any action. However, this mechanism can be used in multi-threaded applications that do not use the SYS/BIOS locking mechanism.

You should not define the functions _lock() and _unlock() functions directly; instead, the installation functions are called to instruct the run-time-support environment to use these new primitives:

```
void _register_lock  (void (  *lock)());
void _register_unlock(void (*unlock)());
```

The arguments to _register_lock() and _register_unlock() should be functions which take no arguments and return no values, and which implement some sort of global semaphore locking:

```
    extern volatile sig_atomic_t *sema = SHARED_SEMAPHORE_LOCATION;
    static int sema_depth = 0;
    static void my_lock(void)
    {
        while (ATOMIC_TEST_AND_SET(sema, MY_UNIQUE_ID) != MY_UNIQUE_ID);
        sema_depth++;
    }
    static void my_unlock(void)
    {
        if (!--sema_depth) ATOMIC_CLEAR(sema);
    }
```

The run-time-support nests calls to _lock(), so the primitives must keep track of the nesting level.

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.

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)
- gmake (GNU make 3.81 or later)

  More information is available from GNU at http://www.gnu.org/software/make. GNU make (gmake) is also available in earlier versions of Code Composer Studio. GNU make is also included in some UNIX support packages for Windows, such as the MKS Toolkit, Cygwin, and Interix. The GNU make used on Windows platforms should explicitly report "This program build for Windows32" when the following is executed from the Command Prompt window:

```
gmake -h
```

All three of these programs are provided as a non-optional feature of CCS 5.1. They are also available as part of the optional XDC Tools feature if you are using an earlier version of CCS.

The mklib program looks for these executables in the following order:

1. in your PATH
2. in the directory getenv("CCS_UTILS_DIR")/cygwin
3. in the directory getenv("CCS_UTILS_DIR")/bin
4. in the directory getenv("XDCROOT")
5. in the directory getenv("XDCROOT")/bin

If you are invoking mklib from the command line, and these executables are not in your path, you must set the environment variable CCS_UTILS_DIR such that getenv("CCS_UTILS_DIR")/bin contains the correct programs.

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 TI_ARM_C_DIR environment variable. Typically, one of the pathnames in TI_ARM_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 TI_ARM_C_DIR to find a library that is the best match for the build attributes of the application. The build attributes are set indirectly according to the command-line options used to build the application. Build attributes include things like CPU revision. If the library name is explicitly specified (e.g. -library=rtsv4_A_be_eabi), run-time support looks for that library exactly. If the library name is not specified, the linker uses the index library libc.a to pick an appropriate library. If the library is specified by path (e.g. –library=/foo/rtsv4_A_be_eabi), it is assumed the library already exists and it will not be built automatically.

The index library describes a set of libraries with different build attributes. The linker will compare the build attributes for each potential library with the build attributes of the application and will pick the best fit. For details on the index library, see the archiver chapter in the *ARM 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 TI_ARM_C_DIR. The library must be in exactly the same directory as the index library libc.a. If the library is not present, the linker invokes mklib to build it. This happens when the library is missing, regardless of whether the user specified the name of the library directly or allowed the linker to pick the best library from the index library.

The mklib program builds the requested library and places it in 'lib' directory part of TI_ARM_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=rtsv4_A_be_eabi.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 version of the library with silicon exception workarounds enabled. The generated library is not a standard library, and must not be placed in the 'lib' directory. It should be placed in a directory local to the project which needs it. To build a

debugging version of the library rtsv4_A_be_eabi, change the working directory to the 'lib' directory and run the command:

```
mklib --pattern=rtsv4_A_be_eabi.lib --name=rtsv4_A_be_eabi_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 7-2.

```
mklib --help
```

Table 7-2. The mklib Program Options

Option	Effect
--index= *filename*	The index library (libc.a) for this release. Used to find a template library for custom builds, and to find the source files (in the lib/src subdirectory of the compiler installation). REQUIRED.
--pattern= *filename*	Pattern for building a library. If neither --extra_options nor --options are specified, the library will be the standard library with the standard options for that library. If either --extra_options or --options are specified, the library is a custom library with custom options. REQUIRED unless --all is used.
--all	Build all standard libraries at once.
--install_to= *directory*	The directory into which to write the library. For a standard library, this defaults to the same directory as the index library (libc.a). For a custom library, this option is REQUIRED.
--compiler_bin_dir= *directory*	The directory where the compiler executables are. When invoking mklib directly, the executables should be in the path, but if they are not, this option must be used to tell mklib where they are. This option is primarily for use when mklib is invoked by the linker.
--name= *filename*	File name for the library with no directory part. Only useful for custom libraries.
--options=' *str* **'**	Options to use when building the library. The default options (see below) are *replaced* by this string. If this option is used, the library will be a custom library.
--extra_options=' *str* **'**	Options to use when building the library. The default options (see below) are also used. If this option is used, the library will be a custom library.
--list_libraries	List the libraries this script is capable of building and exit. ordinary system-specific directory.
--log= *filename*	Save the build log as *filename*.
--tmpdir= *directory*	Use *directory* for scratch space instead of the ordinary system-specific directory.
--gmake= *filename*	Gmake-compatible program to invoke instead of "gmake"
--parallel= *N*	Compile *N* files at once ("gmake -j N").
--query= *filename*	Does this script know how to build FILENAME?
--help or **--h**	Display this help.
--quiet or **--q**	Operate silently.
--verbose or **--v**	Extra information to debug this executable.

Examples:

To build all standard libraries and place them in the compiler's library directory:

```
mklib --all --index=$C_DIR/lib
```

To build one standard library and place it in the compiler's library directory:

```
mklib --pattern=rtsv4_A_be_eabi.lib --index=$C_DIR/lib
```

To build a custom library that is just like rtsv4_A_be_eabi.lib, but has symbolic debugging support enabled:

```
mklib --pattern=rts16.lib --extra_options="-g" --index=$C_DIR/lib --install_to=$Project/Debug
     --name=rtsv4_A_be_eabi_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 that knows how to use the files in the lib/src subdirectory of the compiler installation and invoke gmake with the appropriate options to build each library. If necessary, mklib can be bypassed and the Makefile used directly, but this mode of operation is not supported by TI, and you are responsible for any changes to the Makefile. The format of the Makefile and the interface between mklib and the Makefile is subject to change without notice. The mklib program is the forward-compatible path.

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 TI_ARM_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 7-2 without error, even if they do not do anything.

Chapter 8

C++ Name Demangler

The C++ compiler implements function overloading, operator overloading, and type-safe linking by encoding a function's prototype and namespace in its link-level name. The process of encoding the prototype into the linkname is often referred to as name mangling. When you inspect mangled names, such as in assembly files, disassembler output, or compiler or linker diagnostic messages, it can be difficult to associate a mangled name with its corresponding name in the C++ source code. The C++ name demangler is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code.

These topics tell you how to invoke and use the C++ name demangler. The C++ name demangler reads in input, looking for mangled names. All unmangled text is copied to output unaltered. All mangled names are demangled before being copied to output.

8.1 Invoking the C++ Name Demangler 196
8.2 Sample Usage of the C++ Name Demangler 197

8.1 Invoking the C++ Name Demangler

The syntax for invoking the C++ name demangler is:

armdem [*options*] [*filenames*]

armdem	Command that invokes the C++ name demangler.
options	Options affect how the name demangler behaves. Options can appear anywhere on the command line.
filenames	Text input files, such as the assembly file output by the compiler, the assembler listing file, the disassembly file, and the linker map file. If no filenames are specified on the command line, armdem uses standard input.

By default, the C++ name demangler outputs to standard output. You can use the -o file option if you want to output to a file.

The following options apply only to the C++ name demangler:

--debug (--d)	Prints debug messages.
--diag_wrap[=on,off]	Sets diagnostic messages to wrap at 79 columns (on, which is the default) or not (off).
--help (-h)	Prints a help screen that provides an online summary of the C++ name demangler options.
--output= *file* (-o)	Outputs to the specified file rather than to standard out.
--quiet (-q)	Reduces the number of messages generated during execution.
-u	Specifies that external names do not have a C++ prefix. (deprecated)

8.2 Sample Usage of the C++ Name Demangler

The examples in this section illustrate the demangling process.

This example shows a sample C++ program. In this example, the linknames of all the functions are mangled; that is, their signature information is encoded into their names.

```
class banana {
public:
    int calories(void);
    banana();
    ~banana();
};
int calories_in_a_banana(void)
{
    banana x;
    return x.calories();
}
```

The resulting assembly that is output by the compiler is as follows.

```
_Z20calories_in_a_bananav:
        STMFD     SP!, {A3, A4, V1, LR}
        MOV       A1, SP
        BL        _ZN6bananaC1Ev
        BL        _ZN6banana8caloriesEv
        MOV       V1, A1
        MOV       A1, SP
        BL        _ZN6bananaD1Ev
        MOV       A1, V1
        LDMFD     SP!, {A3, A4, V1, LR}
        BX        LR
```

Executing the C++ name demangler will demangle all names that it believes to be mangled. Enter:

```
armdem calories_in_a_banana.asm
```

The result after running the C++ name demangler is as follows. The linknames in _ZN6bananaC1Ev, _ZN6banana8caloriesEv, and _ZN6bananaD1Ev are demangled.

```
calories_in_a_banana():
        STMFD     SP!, {A3, A4, V1, LR}
        MOV       A1, SP
        BL        banana::banana()
        BL        banana::calories()
        MOV       V1, A1
        MOV       A1, SP
        BL        banana::~banana()
        MOV       A1, V1
        LDMFD     SP!, {A3, A4, V1, LR}
        BX        LR
```

This page intentionally left blank.

Appendix A
Glossary

A.1 Terminology

absolute lister — A debugging tool that allows you to create assembler listings that contain absolute addresses.

alias disambiguation — A technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

aliasing — The ability for a single object to be accessed in more than one way, such as when two pointers point to a single object. It can disrupt optimization, because any indirect reference could refer to any other object.

allocation — A process in which the linker calculates the final memory addresses of output sections.

ANSI — American National Standards Institute; an organization that establishes standards voluntarily followed by industries.

Application Binary Interface (ABI) — A standard that specifies the interface between two object modules. An ABI specifies how functions are called and how information is passed from one program component to another.

archive library — A collection of individual files grouped into a single file by the archiver.

archiver — A software program that collects several individual files into a single file called an archive library. With the archiver, you can add, delete, extract, or replace members of the archive library.

assembler — A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro definitions. The assembler substitutes absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

assignment statement — A statement that initializes a variable with a value.

autoinitialization — The process of initializing global C variables (contained in the .cinit section) before program execution begins.

autoinitialization at run time — An autoinitialization method used by the linker when linking C code. The linker uses this method when you invoke it with the --rom_model link option. The linker loads the .cinit section of data tables into memory, and variables are initialized at run time.

big endian An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

BIS Bit instruction set.

block A set of statements that are grouped together within braces and treated as an entity.

.bss section One of the default object file sections. You use the assembler .bss directive to reserve a specified amount of space in the memory map that you can use later for storing data. The .bss section is uninitialized.

byte Per ANSI/ISO C, the smallest addressable unit that can hold a character.

C/C++ compiler A software program that translates C source statements into assembly language source statements.

code generator A compiler tool that takes the file produced by the parser or the optimizer and produces an assembly language source file.

COFF Common object file format; a system of object files configured according to a standard developed by AT&T. This ABI is no longer supported.

command file A file that contains options, filenames, directives, or commands for the linker or hex conversion utility.

comment A source statement (or portion of a source statement) that documents or improves readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

compiler program A utility that lets you compile, assemble, and optionally link in one step. The compiler runs one or more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

configured memory Memory that the linker has specified for allocation.

constant A type whose value cannot change.

cross-reference listing An output file created by the assembler that lists the symbols that were defined, what line they were defined on, which lines referenced them, and their final values.

.data section One of the default object file sections. The .data section is an initialized section that contains initialized data. You can use the .data directive to assemble code into the .data section.

direct call A function call where one function calls another using the function's name.

directives Special-purpose commands that control the actions and functions of a software tool (as opposed to assembly language instructions, which control the actions of a device).

disambiguation See *alias disambiguation*

dynamic memory allocation A technique used by several functions (such as malloc, calloc, and realloc) to dynamically allocate memory for variables at run time. This is accomplished by

	defining a large memory pool (heap) and using the functions to allocate memory from the heap.
ELF	Executable and Linkable Format; a system of object files configured according to the System V Application Binary Interface specification.
emulator	A hardware development system that duplicates the ARM operation.
entry point	A point in target memory where execution starts.
environment variable	A system symbol that you define and assign to a string. Environmental variables are often included in Windows batch files or UNIX shell scripts such as .cshrc or .profile.
epilog	The portion of code in a function that restores the stack and returns.
executable object file	A linked, executable object file that is downloaded and executed on a target system.
expression	A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.
external symbol	A symbol that is used in the current program module but defined or declared in a different program module.
file-level optimization	A level of optimization where the compiler uses the information that it has about the entire file to optimize your code (as opposed to program-level optimization, where the compiler uses information that it has about the entire program to optimize your code).
function inlining	The process of inserting code for a function at the point of call. This saves the overhead of a function call and allows the optimizer to optimize the function in the context of the surrounding code.
global symbol	A symbol that is either defined in the current module and accessed in another, or accessed in the current module but defined in another.
high-level language debugging	The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.
indirect call	A function call where one function calls another function by giving the address of the called function.
initialization at load time	An autoinitialization method used by the linker when linking C/C++ code. The linker uses this method when you invoke it with the --ram_model link option. This method initializes variables at load time instead of run time.
initialized section	A section from an object file that will be linked into an executable object file.
input section	A section from an object file that will be linked into an executable object file.
integrated preprocessor	A C/C++ preprocessor that is merged with the parser, allowing for faster compilation. Stand-alone preprocessing or preprocessed listing is also available.
interlist feature	A feature that inserts as comments your original C/C++ source statements into the assembly language output from the assembler. The C/C++ statements are inserted next to the equivalent assembly instructions.

intrinsics Operators that are used like functions and produce assembly language code that would otherwise be inexpressible in C, or would take greater time and effort to code.

ISO International Organization for Standardization; a worldwide federation of national standards bodies, which establishes international standards voluntarily followed by industries.

K&R C Kernighan and Ritchie C, the de facto standard as defined in the first edition of *The C Programming Language* (K&R). Most K&R C programs written for earlier, non-ISO C compilers should correctly compile and run without modification.

label A symbol that begins in column 1 of an assembler source statement and corresponds to the address of that statement. A label is the only assembler statement that can begin in column 1.

linker A software program that combines object files to form an executable object file that can be allocated into system memory and executed by the device.

listing file An output file, created by the assembler, which lists source statements, their line numbers, and their effects on the section program counter (SPC).

little endian An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

loader A device that places an executable object file into system memory.

loop unrolling An optimization that expands small loops so that each iteration of the loop appears in your code. Although loop unrolling increases code size, it can improve the performance of your code.

macro A user-defined routine that can be used as an instruction.

macro call The process of invoking a macro.

macro definition A block of source statements that define the name and the code that make up a macro.

macro expansion The process of inserting source statements into your code in place of a macro call.

map file An output file, created by the linker, which shows the memory configuration, section composition, section allocation, symbol definitions and the addresses at which the symbols were defined for your program.

memory map A map of target system memory space that is partitioned into functional blocks.

name mangling A compiler-specific feature that encodes a function name with information regarding the function's arguments return types.

object file An assembled or linked file that contains machine-language object code.

object library An archive library made up of individual object files.

operand An argument of an assembly language instruction, assembler directive, or macro directive that supplies information to the operation performed by the instruction or directive.

optimizer A software tool that improves the execution speed and reduces the size of C programs.

options Command-line parameters that allow you to request additional or specific functions when you invoke a software tool.

output section A final, allocated section in a linked, executable module.

parser A software tool that reads the source file, performs preprocessing functions, checks the syntax, and produces an intermediate file used as input for the optimizer or code generator.

partitioning The process of assigning a data path to each instruction.

pop An operation that retrieves a data object from a stack.

pragma A preprocessor directive that provides directions to the compiler about how to treat a particular statement.

preprocessor A software tool that interprets macro definitions, expands macros, interprets header files, interprets conditional compilation, and acts upon preprocessor directives.

program-level optimization An aggressive level of optimization where all of the source files are compiled into one intermediate file. Because the compiler can see the entire program, several optimizations are performed with program-level optimization that are rarely applied during file-level optimization.

prolog The portion of code in a function that sets up the stack.

push An operation that places a data object on a stack for temporary storage.

quiet run An option that suppresses the normal banner and the progress information.

raw data Executable code or initialized data in an output section.

relocation A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

run-time environment The run time parameters in which your program must function. These parameters are defined by the memory and register conventions, stack organization, function call conventions, and system initialization.

run-time-support functions Standard ISO functions that perform tasks that are not part of the C language (such as memory allocation, string conversion, and string searches).

run-time-support library A library file, rts.src, which contains the source for the run time-support functions.

section A relocatable block of code or data that ultimately will be contiguous with other sections in the memory map.

sign extend	A process that fills the unused MSBs of a value with the value's sign bit.
simulator	A software development system that simulates ARM 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 an emulator or simulator.
target system	The system on which the object code you have developed is executed.
.text section	One of the default object file sections. The .text section is initialized and contains executable code. You can use the .text directive to assemble code into the .text section.
trigraph sequence	A 3-character sequence that has a meaning (as defined by the ISO 646-1983 Invariant Code Set). These characters cannot be represented in the C character set and are expanded to one character. For example, the trigraph ??' is expanded to ^.
trip count	The number of times that a loop executes before it terminates.
unconfigured memory	Memory that is not defined as part of the memory map and cannot be loaded with code or data.
uninitialized section	A object file section that reserves space in the memory map but that has no actual contents. These sections are built with the .bss and .usect directives.
unsigned value	A value that is treated as a nonnegative number, regardless of its actual sign.
variable	A symbol representing a quantity that can assume any of a set of values.

veneer A sequence of instructions that serves as an alternate entry point into a routine if a state change is required.

word A 32-bit addressable location in target memory

Revision History

Changes from March 11, 2020 to March 31, 2023 (from Revision V (March 2020) to Revision W (March 2023)) **Page**

- Updated the numbering format for tables, figures, and cross-references throughout the document.... 9
- Removed references to the Processors wiki throughout the document.... 9
- The --strict_compatibility linker option no longer has any effect and has been removed from the documentation.... 25
- Documented predefined macros for ptrdiff_t and size_t types.... 35
- Corrected names of the --gen_cross_reference_listing and --asm_cross_reference_listing options wherever they appear.... 45
- Clarified that --opt_level=4 must be placed before --run_linker option.... 59
- Corrected information about default for --gen_data_subsections option and its interaction with the SET_DATA_SECTION pragma.... 76
- Updated information about the size of enum types.... 91
- Removed documentation for the CODE_ALIGN pragma, which is not supported. Use the aligned function attribute instead.... 98
- Clarify interaction between --opt_level and FUNCTION_OPTIONS pragma.... 105
- Documented C++ attribute syntax for attributes that correspond to the MUST_ITERATE pragma.... 107
- Added documentation for the PROB_ITERATE pragma.... 112
- Documented C++ attribute syntax for attributes that correspond to the UNROLL pragma.... 116
- Added example using the location attribute.... 133
- Clarified information about string handling functions.... 175
- Added information about time and clock RTS functions.... 176

The following table lists changes made to this document prior to changes to the document numbering format. The left column identifies the first version of this document in which that particular change appeared.

Earlier Revisions

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPNU151V	Linking	Section 4.3.5	Clarified that either --rom_model or --ram_model is required if only the linker is being run, but --rom_model is the default if the compiler runs on C/C++ files on the same command line.
SPNU151V	C/C++ Language	Section 5.11.22	The `#pragma once` is now documented for use in header files.
SPNU151V	Run-Time Environment	Section 6.10.3.1	Clarified that zero initialization takes place only if the --rom_model linker option is used, not if the --ram_model option is used.
SPNU151U		-- throughout --	The default file extensions for object files created by the compiler have been changed in order to prevent conflicts when C and C++ files have the same names. Object files generated from C source files have the .c.obj extension. Object files generated from C++ source files have the .cpp.obj extension.
SPNU151T	Using the Compiler	Section 2.3.1	Added the --emit_references:file linker option.
SPNU151T	Using the Compiler	Section 2.5.1	Documented that C standard macros such as __STDC_VERSION__ are supported.
SPNU151T	C/C++ Language	Section 5.11	Added documentation for the CODE_ALIGN pragma.
SPNU151T	C/C++ Language	Section 5.11.19	Clarify section placement for the NOINIT and PERSISTENT pragmas.

Earlier Revisions (continued)

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPNU151T	C/C++ Language	Section 5.14	Corrected syntax for the _norm intrinsic.
SPNU151T	C/C++ Language	Section 5.16.1	Updated list of C99 non-supported run-time functions.
SPNU151T	C/C++ Language	Section 5.17.2	Added documentation for the aligned, calls, naked, and weak function attributes.
SPNU151T	C/C++ Language	Section 5.17.4	Added documentation for the location and packed variable attributes.
SPNU151T	Run-Time Support Functions	DEV_lseek topic	Corrected syntax documented for DEV_lseek function.
SPNU151S	Introduction, Using the Compiler, C/C++ Language	Section 1.3, Section 2.3, Section 5.1, and Section 5.16.2	Added support for C11.
SPNU151S	Using the Compiler	Section 2.3.1	Added the --ecc=on linker option, which enables ECC generation. Note that ECC generation is now off by default.
SPNU151S	Using the Compiler	Section 2.5.1	The __TI_STRICT_ANSI_MODE__ and __TI_STRICT_FP_MODE__ macros are defined as 0 if their conditions are false.
SPNU151S	Using the Compiler, C/C++ Language	Section 2.11 and Section 5.11	Revised the section on inline function expansion and its subsections to include new pragmas and changes to the compilers decision-making about what functions to inline. The FORCEINLINE, FORCEINLINE_RECURSIVE, and NOINLINE pragmas have been added.
SPNU151S	C/C++ Language	Section 5.2	C++11 features related to atomics are now supported. In addition, removed several C++ features from the exception list because they have been supported for several releases.
SPNU151S	C/C++ Language	Section 5.6	Added information about character sets and file encoding.
SPNU151S	C/C++ Language	Section 5.14	Corrected syntax for _smac intrinsic.
SPNU151S	C/C++ Language	Section 5.17.2 and Section 5.17.4	Added "retain" as a function attribute and variable attribute.
SPNU151S	C/C++ Language	Section 5.17.6	Clarified the availability of the __builtin_sqrt() and __builtin_sqrtf() functions.
SPNU151R	Using the Compiler, C/C++ Language	Section 2.3 and Section 5.2	The compiler now follows the C++14 standard.
SPNU151R	C/C++ Language	Section 5.17	The compiler now supports several Clang __has_ macro extensions.
SPNU151R	C/C++ Language	Section 5.17.1	The wrapper header file GCC extension (#include_next) is now supported.
SPNU151Q	Using the Compiler, C/C++ Language	Table 2-31, Section 5.1, Section 5.14, Section 5.17.2	ARM C Language Extensions (ACLE) are supported.
SPNU151Q	Using the Compiler	Section 2.14	Updated the list of settings for the --float_support option.
SPNU151Q	C/C++ Language	Section 5.2	Preliminary changes have been made in order to support C++14 in a future release. These changes may cause linktime errors. Recompile object files to resolve these errors.
SPNU151Q	C/C++ Language	Section 5.7.1	Clarified exceptions to const data storage set by the const keyword.
SPNU151Q	C/C++ Language	Section 5.14	Remove incorrect third parameter for the _smuad, _smuadx, _smusd, and _smusdx intrinsics.
SPNU151P	Optimization	Section 3.7.1.4	Corrected error in command to process the profile data.
SPNU151O	Using the Compiler, C/C++ Language	Section 2.3.3	Revised to state that --check_misra option is required even if the CHECK_MISRA pragma is used.
SPNU151O	Using the Compiler, C/C++ Language, and Run-Time Support Functions	Section 2.5.1, Section 5.16, and Section 7.1.1	_AEABI_PORTABILITY_LEVEL can be defined to enable full object file portability when headers files are included.
SPNU151O	Using the Compiler	Section 2.10	Corrected the document to describe the ---gen_preprocessor_listing option. The name --gen_parser_listing was incorrect.
SPNU151N	Optimization	Section 3.7.3	Corrected function names for _TI_start_pprof_collection() and _TI_stop_pprof_collection().

Earlier Revisions (continued)

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPNU151M	Using the Compiler	Section 2.3	The default for --cinit_compression and --copy_compression has been changed from RLE to LZSS.
SPNU151M	Using the Compiler	--	Several compiler options have been deprecated, removed, or renamed. The compiler continues to accept some of the deprecated options, but they are not recommended for use.
SPNU151M	Using the Compiler	Section 2.5.1	The __little_endian__ and __big_endian__ macros are preceded by two underscores.
SPNU151M	C/C++ Language	Section 5.14	The following intrinsics are supported for Cortex-M3: __ldrex, __ldrexb, __ldrexh, __strex, __strexb, and __strexh.
SPNU151M	Run-Time Environment	Section 6.8.1	The _enable_interrupts, _enable_IRQ, _enable_FIQ, _disable_interrupts, _disable_IRQ, and _disable_FIQ intrinsics for Cortex-R4 and Cortex-A8 now use the CPSIE and CPSID instructions.
SPNU151L	Using the Compiler	Section 2.3 and Section 4.2.2	The --gen_data_subsections option has been added.
SPNU151L	Using the Compiler	Section 2.3.5	The --symdebug:dwarf_version option can be set to 4 to enable the use of DWARF debugging format version 4.
SPNU151L	Optimization	Section 3.7 and Section 3.8	Feedback directed optimization is described. This technique can be used for code coverage analysis.
SPNU151L	C/C++ Language	Section 5.11.1	A CALLS pragma has been added to specify a set of functions that can be called indirectly from a specified calling function. Using this pragma allows such indirect calls to be included in the calculation of a functions' inclusive stack size.
SPNU151L	C/C++ Language	Section 5.14	The following intrinsics have been added to the documentation: __MCR, __MRC.
SPNU151L	Run-Time Environment	Section 6.10.1	Additional boot hook functions are available. These can be customized for use during system initialization.
SPNU151K	Introduction	Section 1.4	The COFF object file format and the TI_ARM9_ABI and TIARM ABIs are no longer supported. The ARM Code Generation Tools now support only the Embedded Application Binary Interface (EABI) ABI, which works only with object files that use the ELF object file format and the DWARF debug format. Sections of this document that referred to the COFF format have been removed or simplified. If you would like to produce COFF output files, please use v5.2 of the ARM Code Generation Tools and refer to SPNU151J for documentation. The --abi=coff, --symdebug:profile_coff, --no_sym_merge, and --diable_clink options have been deprecated.
SPNU151K	Using the Compiler	Section 2.3.4	The --ramfunc option has been added. If set, this option places all functions in RAM.
SPNU151K	C/C++ Language	Section 5.14	The following intrinsics have been added to the documentation: __nop, __sqrt, __sqrtf, __wfi, __wfe
SPNU151K	C/C++ Language	Section 5.17.2	The ramfunc function attribute has been added. It specifies that a function should be placed in RAM.
SPNU151K	Run-Time Support Functions	Section 7.1.2	Added information about header file extensions.
SPNU151J	Introduction	Section 1.3	Added support for C99 and C++03.
SPNU151J	Using the Compiler	Table 2-1	Added --endian=[big | little] option.
SPNU151J	Using the Compiler	Table 2-6, Section 2.7, and Section 2.3.3	Added the --advice:power and --advice:power_severity options for use with the ULP Advisor.
SPNU151J	Using the Compiler	Table 2-8	Added support for C99 and C++03. The -gcc option has been deprecated. The --relaxed_ansi option is now the default.
SPNU151J	Using the Compiler	Table 2-8	Removed documentation of precompiled headers, which have been deprecated.
SPNU151J	Using the Compiler	Table 2-11 and Section 2.7.1	Added --section_sizes option for diagnostic reporting of section sizes.

Earlier Revisions (continued)

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPNU151J	Using the Compiler	Table 2-28 and Section 4.3.3	Added the –cinit_hold_wdt linker option.
SPNU151J	Using the Compiler	Section 2.5.1	Added __TI_ ARM_V7M4__ predefined macro name for Cortex-M4.
SPNU151J	Using the Compiler	Section 2.5.3	Documented that the #warning and #warn preprocessor directives are supported.
SPNU151J	Using the Compiler	Section 2.6	Added section on techniques for passing arguments to main().
SPNU151J	Using the Compiler	Section 2.11	Documented that the inline keyword is now enabled in all modes except C89 strict ANSI mode.
SPNU151J	C/C++ Language	Section 5.1.1	Added section documenting implementation-defined behavior.
SPNU151J	C/C++ Language	Section 5.4	Added support for the ULP Advisor
SPNU151J	C/C++ Language	Section 5.5.1	Added documentation on the size of enum types.
SPNU151J	C/C++ Language	Section 5.11.3, Section 5.11.12, Section 5.11.13, Section 5.11.19, and Section 5.11.26	Added the CHECK_ULP, FUNC_ALWAYS_INLINE, FUNC_CANNOT_INLINE, NOINIT, PERSISTENT, and RESET_ULP pragmas.
SPNU151J	C/C++ Language	Section 5.11.16, Section 5.11.27, and Section 5.17.2	Added C++ syntax for the INTERRUPT and RETAIN pragmas. Also removed unnecessary semicolons from #pragma syntax specifications. Also the GCC interrupt and alias function attributes are now supported.
SPNU151J	C/C++ Language	Section 5.11.8	Added the diag_push and diag_pop diagnostic message pragmas.
SPNU151J	C/C++ Language	Section 5.14	Added __delay_cycles, __get_PRIMASK, __set_PRIMASK, __get_MSP, and __set_MSP intrinsics.
SPNU151J	C/C++ Language	Section 5.14	Corrected arguments for smlalbb, smlalbt, smlaltb, smlaltt, smlabb, smlabt, smlatb, and smlatt intrinsics.
SPNU151J	C/C++ Language	Section 5.16, Section 5.16.1, and Section 5.16.3	Added support for C99 and C++03. The --relaxed_ansi option is now the default and --strict_ansi is the other option; "normal mode" for standards violation strictness is no longer available.
SPNU151J	Run-Time Environment	Section 6.5	Added reference to section on accessing linker symbols in C and C++ in the *Assembly Language Tools User's Guide*.
SPNU151J	Run-Time Environment	Section 6.7.5	Added information about allowable return values from SWI handlers.
SPNU151J	Run-Time Environment	Section 6.8.1	Added instructions for several device families for _disable_interrupts, _enable_interrupts, and _restore_interrupts intrinsics. Added Cortex-M support for _enable_IRQ, _disable_IRQ, and _set_interrupt_priority intrinsics.
SPNU151J	Run-Time Environment	Section 6.10.1	Added support for system pre-initialization.
SPNU151J	Run-Time Support Functions	Section 7.1.3	RTS source code is no longer provided in a rtssrc.zip file. Instead, it is located in separate files in the lib/src subdirectory of the compiler installation.
SPNU151J	C++ Name Demangler	Section 8.1	Corrected information about name demangler options.
SPNU151J	C++ Name Demangler	Section 8.2	Corrected examples of resulting assembly output.

This page intentionally left blank.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265