ARM Optimizing C/C++ Compiler v5.0

User's Guide

Literature Number: SPNU151H
August 2012

Texas Instruments

Contents

Preface 9

1 Introduction to the Software Development Tools 12
- 1.1 Software Development Tools Overview 13
- 1.2 C/C++ Compiler Overview 14
  - 1.2.1 ANSI/ISO Standard 14
  - 1.2.2 Output Files 15
  - 1.2.3 Compiler Interface 15
  - 1.2.4 Utilities 15

2 Using the C/C++ Compiler 16
- 2.1 About the Compiler 17
- 2.2 Invoking the C/C++ Compiler 17
- 2.3 Changing the Compiler's Behavior With Options 18
  - 2.3.1 Linker Options 24
  - 2.3.2 Frequently Used Options 26
  - 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 32
  - 2.3.7 Changing How the Compiler Interprets Filenames 33
  - 2.3.8 Changing How the Compiler Processes C Files 33
  - 2.3.9 Changing How the Compiler Interprets and Names Extensions 33
  - 2.3.10 Specifying Directories 33
  - 2.3.11 Assembler Options 34
  - 2.3.12 Deprecated Options 35
- 2.4 Controlling the Compiler Through Environment Variables 35
  - 2.4.1 Setting Default Compiler Options (TI_ARM_C_OPTION) 35
  - 2.4.2 Naming an Alternate Directory (TI_ARM_C_DIR) 36
- 2.5 Precompiled Header Support 37
  - 2.5.1 Automatic Precompiled Header 37
  - 2.5.2 Manual Precompiled Header 37
  - 2.5.3 Additional Precompiled Header Options 37
- 2.6 Controlling the Preprocessor 38
  - 2.6.1 Predefined Macro Names 38
  - 2.6.2 The Search Path for #include Files 39
  - 2.6.3 Generating a Preprocessed Listing File (--preproc_only Option) 40
  - 2.6.4 Continuing Compilation After Preprocessing (--preproc_with_compile Option) 41
  - 2.6.5 Generating a Preprocessed Listing File With Comments (--preproc_with_comment Option) 41
  - 2.6.6 Generating a Preprocessed Listing File With Line-Control Information (--preproc_with_line Option) 41
  - 2.6.7 Generating Preprocessed Output for a Make Utility (--preproc_dependency Option) 41
  - 2.6.8 Generating a List of Files Included With the #include Directive (--preproc_includes Option) 41
  - 2.6.9 Generating a List of Macros in a File (--preproc_macros Option) 41
- 2.7 Understanding Diagnostic Messages 41
  - 2.7.1 Controlling Diagnostics 43
  - 2.7.2 How You Can Use Diagnostic Suppression Options 43

Copyright © 2012, Texas Instruments Incorporated

2.8 Other Messages ... 44
2.9 Generating Cross-Reference Listing Information (--gen_acp_xref Option) ... 44
2.10 Generating a Raw Listing File (--gen_acp_raw Option) ... 45
2.11 Using Inline Function Expansion ... 46
2.11.1 Inlining Intrinsic Operators ... 46
2.11.2 Using the inline Keyword, the --no_inlining Option, and Level 3 Optimization ... 46
2.11.3 Automatic Inlining ... 46
2.11.4 Inlining Restrictions ... 47
2.12 Using Interlist ... 47
2.13 Controlling Application Binary Interface ... 48
2.14 VFP Support ... 49
2.15 Enabling Entry Hook and Exit Hook Functions ... 50
3 Optimizing Your Code ... 51
3.1 Invoking Optimization ... 52
3.2 Performing File-Level Optimization (--opt_level=3 option) ... 53
3.2.1 Controlling File-Level Optimization (--std_lib_func_def Options) ... 53
3.2.2 Creating an Optimization Information File (--gen_opt_info Option) ... 53
3.3 Performing Program-Level Optimization (--program_level_compile and --opt_level=3 options) ... 54
3.3.1 Controlling Program-Level Optimization (--call_assumptions Option) ... 54
3.3.2 Optimization Considerations When Mixing C/C++ and Assembly ... 55
3.4 Link-Time Optimization (--opt_level=4 Option) ... 56
3.4.1 Option Handling ... 56
3.4.2 Incompatible Types ... 57
3.5 Accessing Aliased Variables in Optimized Code ... 57
3.6 Use Caution With asm Statements in Optimized Code ... 57
3.7 Automatic Inline Expansion (--auto_inline Option) ... 57
3.8 Using the Interlist Feature With Optimization ... 58
3.9 Debugging and Profiling Optimized Code ... 60
3.9.1 Debugging Optimized Code (--symdebug:dwarf, --symdebug:coff, and --opt_level Options) ... 60
3.9.2 Profiling Optimized Code ... 60
3.10 Controlling Code Size Versus Speed ... 61
3.11 What Kind of Optimization Is Being Performed? ... 62
3.11.1 Cost-Based Register Allocation ... 62
3.11.2 Alias Disambiguation ... 62
3.11.3 Branch Optimizations and Control-Flow Simplification ... 63
3.11.4 Data Flow Optimizations ... 63
3.11.5 Expression Simplification ... 63
3.11.6 Inline Expansion of Functions ... 63
3.11.7 Function Symbol Aliasing ... 63
3.11.8 Induction Variables and Strength Reduction ... 64
3.11.9 Loop-Invariant Code Motion ... 64
3.11.10 Loop Rotation ... 64
3.11.11 Instruction Scheduling ... 64
3.11.12 Tail Merging ... 64
3.11.13 Autoincrement Addressing ... 64
3.11.14 Block Conditionalizing ... 65
3.11.15 Epilog Inlining ... 65
3.11.16 Removing Comparisons to Zero ... 65
3.11.17 Integer Division With Constant Divisor ... 65
3.11.18 Branch Chaining ... 66
4 Linking C/C++ Code ... 67
4.1 Invoking the Linker Through the Compiler (-z Option) ... 68
4.1.1 Invoking the Linker Separately ... 68

Copyright © 2012, Texas Instruments Incorporated

4.1.2 Invoking the Linker as Part of the Compile Step 69
4.1.3 Disabling the Linker (--compile_only Compiler Option) 69
4.2 Linker Code Optimizations 70
4.2.1 Generate List of Dead Functions (--generate_dead_funcs_list Option) 70
4.2.2 Generating Function Subsections (--gen_func_subsections Compiler Option) 70
4.3 Controlling the Linking Process 71
4.3.1 Including the Run-Time-Support Library 71
4.3.2 Run-Time Initialization 72
4.3.3 Global Object Constructors 72
4.3.4 Specifying the Type of Global Variable Initialization 73
4.3.5 Specifying Where to Allocate Sections in Memory 74
4.3.6 A Sample Linker Command File 75

5 ARM C/C++ Language Implementation 76

5.1 Characteristics of ARM C 77
5.2 Characteristics of ARM C++ 77
5.3 Using MISRA-C:2004 78
5.4 Data Types 79
5.5 Keywords 80
5.5.1 The const Keyword 80
5.5.2 The interrupt Keyword 80
5.5.3 The volatile Keyword 81
5.6 C++ Exception Handling 82
5.7 Register Variables and Parameters 83
5.7.1 Local Register Variables and Parameters 83
5.7.2 Global Register Variables 83
5.8 The asm Statement 84
5.9 Pragma Directives 85
5.9.1 The CHECK_MISRA Pragma 86
5.9.2 The CLINK Pragma 86
5.9.3 The CODE_SECTION Pragma 86
5.9.4 The CODE_STATE Pragma 87
5.9.5 The DATA_ALIGN Pragma 87
5.9.6 The DATA_SECTION Pragma 88
5.9.7 The Diagnostic Message Pragmas 89
5.9.8 The DUAL_STATE Pragma 89
5.9.9 The FUNC_EXT_CALLED Pragma 90
5.9.10 The FUNCTION_OPTIONS Pragma 90
5.9.11 The INTERRUPT Pragma 90
5.9.12 The LOCATION Pragma 91
5.9.13 The MUST_ITERATE Pragma 91
5.9.14 The NO_HOOKS Pragma 93
5.9.15 The RESET_MISRA Pragma 93
5.9.16 The RETAIN Pragma 93
5.9.17 The SET_CODE_SECTION and SET_DATA_SECTION Pragmas 94
5.9.18 The SWI_ALIAS Pragma 95
5.9.19 The TASK Pragma 96
5.9.20 The UNROLL Pragma 96
5.9.21 The WEAK Pragma 97
5.10 The _Pragma Operator 97
5.11 Application Binary Interface 98
5.11.1 TI_ARM9_ABI 98
5.11.2 TIABI (Deprecated) 98
5.11.3 ARM ABIv2 or EABI 99

Copyright © 2012, Texas Instruments Incorporated

5.12 ARM Instruction Intrinsics 99
5.13 Object File Symbol Naming Conventions (Linknames) 106
5.14 Initializing Static and Global Variables in TI ARM9 ABI and TIABI Modes 107
5.14.1 Initializing Static and Global Variables With the Linker 107
5.14.2 Initializing Static and Global Variables With the const Type Qualifier 107
5.15 Changing the ANSI/ISO C Language Mode 108
5.15.1 Compatibility With K&R C (--kr_compatible Option) 108
5.15.2 Enabling Strict ANSI/ISO Mode and Relaxed ANSI/ISO Mode (--strict_ansi and --relaxed_ansi Options) 109
5.15.3 Enabling Embedded C++ Mode (--embedded_cpp Option) 109
5.16 GNU Language Extensions 110
5.16.1 Extensions 110
5.16.2 Function Attributes 111
5.16.3 Variable Attributes 111
5.16.4 Type Attributes 112
5.16.5 Built-In Functions 112
5.17 AUTOSAR 113
5.18 Compiler Limits 113

6 Run-Time Environment 114

6.1 Memory Model 115
6.1.1 Sections 115
6.1.2 C/C++ System Stack 116
6.1.3 Dynamic Memory Allocation 117
6.1.4 Initialization of Variables in TI_ARM9_ABI and TIABI 117
6.2 Object Representation 117
6.2.1 Data Type Storage 117
6.2.2 Bit Fields 121
6.2.3 Character String Constants 123
6.3 Register Conventions 124
6.4 Function Structure and Calling Conventions 127
6.4.1 How a Function Makes a Call 128
6.4.2 How a Called Function Responds 129
6.4.3 C Exception Handler Calling Convention 129
6.4.4 Accessing Arguments and Local Variables 130
6.4.5 Generating Long Calls (-ml Option) in 16-bit Mode 130
6.5 Interfacing C and C++ With Assembly Language 130
6.5.1 Using Assembly Language Modules With C/C++ Code 130
6.5.2 Accessing Assembly Language Variables From C/C++ 132
6.5.3 Sharing C/C++ Header Files With Assembly Source 133
6.5.4 Using Inline Assembly Language 133
6.5.5 Modifying Compiler Output 133
6.6 Interrupt Handling 134
6.6.1 Saving Registers During Interrupts 134
6.6.2 Using C/C++ Interrupt Routines 134
6.6.3 Using Assembly Language Interrupt Routines 134
6.6.4 How to Map Interrupt Routines to Interrupt Vectors 135
6.6.5 Using Software Interrupts 136
6.6.6 Other Interrupt Information 136
6.7 Intrinsic Run-Time-Support Arithmetic and Conversion Routines 137
6.7.1 Naming Conventions 137
6.7.2 CPSR Register and Interrupt Intrinsics 139
6.8 Built-In Functions 140
6.9 System Initialization 140

Copyright © 2012, Texas Instruments Incorporated

6.9.1 Run-Time Stack 141
6.9.2 TI ARM9 ABI/TIABI Automatic Initialization of Variables 141
6.9.3 EABI Automatic Initialization of Variables 143
6.9.4 Initialization Tables 148
6.10 Dual-State Interworking Under TIABI (Deprecated) 151
6.10.1 Level of Dual-State Support 151
6.10.2 Implementation 152
7 Using Run-Time-Support Functions and Building Libraries 155
7.1 C and C++ Run-Time Support Libraries 156
7.1.1 Linking Code With the Object Library 156
7.1.2 Header Files 156
7.1.3 Modifying a Library Function 156
7.1.4 Minimal Support for Internationalization 157
7.1.5 Allowable Number of Open Files 157
7.1.6 Nonstandard Header Files in rtssrc.zip 157
7.1.7 Library Naming Conventions 158
7.2 The C I/O Functions 159
7.2.1 High-Level I/O Functions 159
7.2.2 Overview of Low-Level I/O Implementation 160
7.2.3 Device-Driver Level I/O Functions 164
7.2.4 Adding a User-Defined Device Driver for C I/O 168
7.2.5 The device Prefix 169
7.3 Handling Reentrancy (_register_lock() and _register_unlock() Functions) 171
7.4 Library-Build Process 172
7.4.1 Required Non-Texas Instruments Software 172
7.4.2 Using the Library-Build Process 172
7.4.3 Extending mklib 175
8 C++ Name Demangler 176
8.1 Invoking the C++ Name Demangler 177
8.2 C++ Name Demangler Options 177
8.3 Sample Usage of the C++ Name Demangler 178
9 Static Stack Depth Profiler 180
9.1 Invoking the Static Stack Depth Profiler 181
9.2 Stack Depth Statistics Listing 181
9.3 Dependencies and Limitations 183
9.4 User Input Mechanisms 183
9.5 Configuration File Specification 184
9.5.1 Specify Indirect Calls 184
9.5.2 Specifying Reentrant Procedures 185
9.5.3 Specifying Interrupt Service Routines 185
9.5.4 Other Features 186
9.5.5 Tail Calls 186
9.6 Profiler Generated Warnings 187
9.7 Run-Time Support for Stack Depth Profiling 187
A Glossary 188

Copyright © 2012, Texas Instruments Incorporated

List of Figures

1-1. ARM Software Development Flow 13
6-1. Char and Short Data Storage Format 118
6-2. 32-Bit Data Storage Format 119
6-3. Double-Precision Floating-Point Data Storage Format 120
6-4. Bit-Field Packing in Big-Endian and Little-Endian Formats 121
6-5. Use of the Stack During a Function Call 128
6-6. Autoinitialization at Run Time 142
6-7. Initialization at Load Time 142
6-8. Autoinitialization at Run Time in EABI Mode 144
6-9. Initialization at Load Time in EABI Mode 148
6-10. Constructor Table for EABI Mode 148
6-11. Format of Initialization Records in the .cinit Section 149
6-12. Format of Initialization Records in the .pinit Section 150
9-1. Application Call Trees 181

List of Tables

2-1. Processor Options 18
2-2. Optimization Options 18
2-3. Advanced Optimization Options 19
2-4. Debug Options 19
2-5. Advanced Debug Options 19
2-6. Include Options 19
2-7. Control Options 20
2-8. Language Options 20
2-9. Parser Preprocessing Options 20
2-10. Predefined Symbols Options 21
2-11. Diagnostics Options 21
2-12. Run-Time Model Options 22
2-13. Entry/Exit Hook Options 22
2-14. Library Function Assumptions Options 22
2-15. Assembler Options 22
2-16. File Type Specifier Options 23
2-17. Directory Specifier Options 23
2-18. Default File Extensions Options 24
2-19. Command Files Options 24
2-20. MISRA-C:2004 Options 24
2-21. Linker Basic Options 24
2-22. File Search Path Options 24
2-23. Command File Preprocessing Options 25
2-24. Diagnostic Options 25
2-25. Linker Output Options 25
2-26. Symbol Management Options 25
2-27. Run-Time Environment Options 26
2-28. Link-Time Optimization Options 26
2-29. Miscellaneous Options 26
2-30. Compiler Backwards-Compatibility Options Summary 35
2-31. Predefined ARM Macro Names 38

Copyright © 2012, Texas Instruments Incorporated

2-32. Raw Listing File Identifiers 45
2-33. Raw Listing File Diagnostic Identifiers 45
3-1. Options That You Can Use With --opt_level=3 53
3-2. Selecting a File-Level Optimization Option 53
3-3. Selecting a Level for the --gen_opt_info Option 53
3-4. Selecting a Level for the --call_assumptions Option 54
3-5. Special Considerations When Using the --call_assumptions Option 55
4-1. Initialized Sections Created by the Compiler for TI ARM9 ABI and TIABI (deprecated) 74
4-2. Initialized Sections Created by the Compiler for EABI 74
4-3. Uninitialized Sections Created by the Compiler 74
5-1. ARM C/C++ Data Types 79
5-2. EABI Enumerator Types 79
5-3. ARM Intrinsic Support by Target 99
5-4. ARM Compiler Intrinsics 101
5-5. GCC Language Extensions 110
6-1. Summary of Sections and Memory Placement 116
6-2. Data Representation in Registers and Memory 117
6-3. How Register Types Are Affected by the Conventions 124
6-4. Register Usage 124
6-5. VFP Register Usage 125
6-6. Neon Register Usage 126
6-7. Naming Convention Prefixes 137
6-8. Summary of Run-Time-Support Arithmetic and Conversion Functions 137
6-9. Run-Time-Support Function Register Usage Conventions 138
6-10. CPSR and Interrupt C/C++ Compiler Intrinsics 139
6-11. Selecting a Level of Dual-State Support 151
7-1. The mklib Program Options 174

Copyright © 2012, Texas Instruments Incorporated

Read This First

About This Manual

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

- Compiler
- Library build utility
- C++ name demangler
- Static stack depth profiler

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

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

Notational Conventions

This document uses the following conventions:

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

  Here is a sample of C code:

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

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

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*

Copyright © 2012, Texas Instruments Incorporated

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

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

- Some directives can have a varying number of parameters. For example, the .byte directive. This syntax is shown as [, ..., *parameter*].
- The 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 C Standard), International Organization for Standardization

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

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

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

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

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

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

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

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

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

Copyright © 2012, Texas Instruments Incorporated

Related Documentation From Texas Instruments

You can use the following books 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.

ARM is a registered trademark of ARM Limited.
All other trademarks are the property of their respective owners.

Copyright © 2012, Texas Instruments Incorporated

Chapter 1
SPNU151H–August 2012

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*.

Topic **Page**

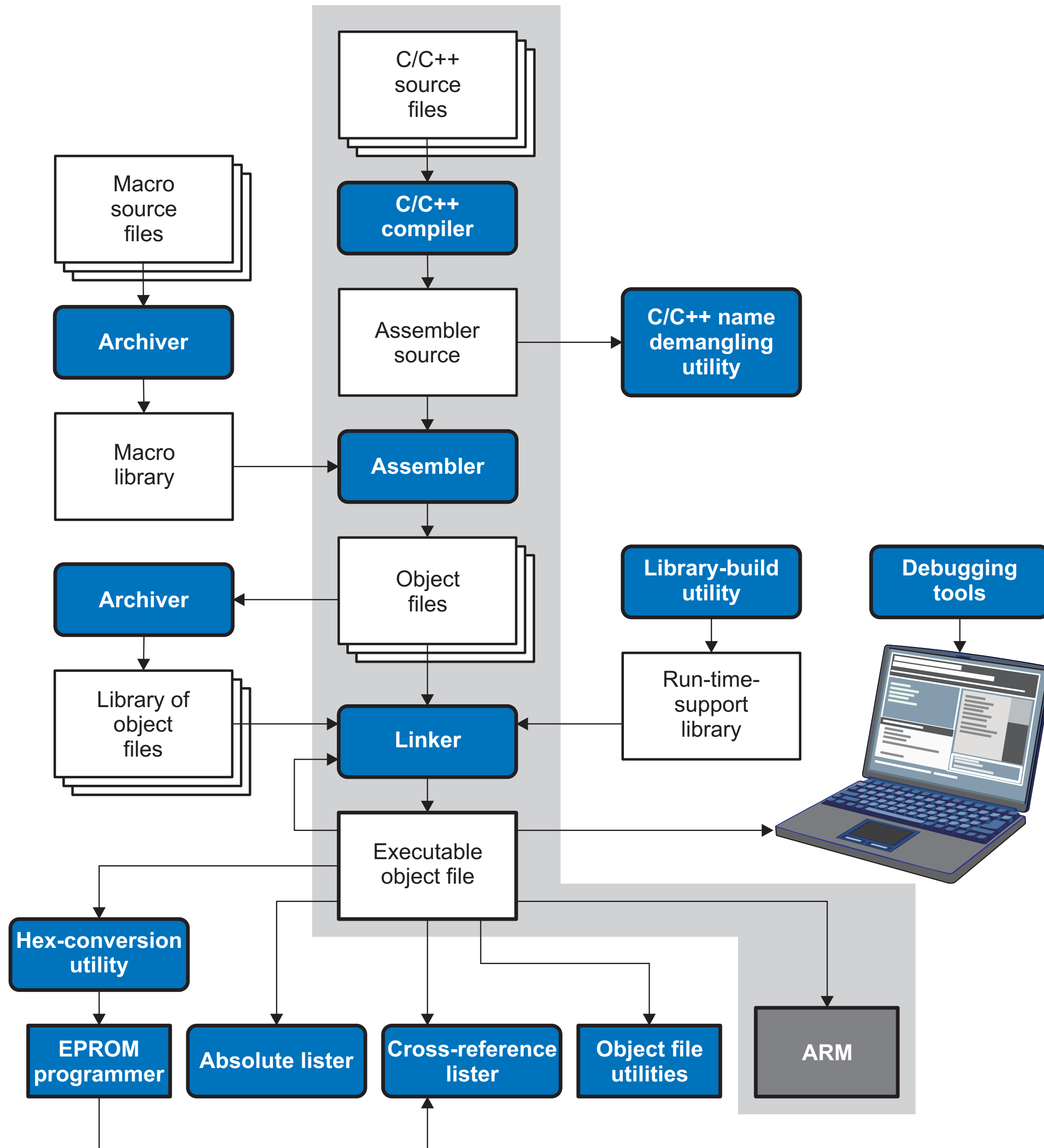
1.1 Software Development Tools Overview **13**
1.2 C/C++ Compiler Overview **14**

Copyright © 2012, Texas Instruments Incorporated

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



Copyright © 2012, Texas Instruments Incorporated

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. The *ARM Assembly Language Tools User's Guide* explains how to use the assembler.
- The **linker** combines relocatable object files into a single absolute executable object file. As it creates the executable file, it performs relocation and resolves external references. The linker accepts relocatable object files and object libraries as input. See Chapter 4. The *ARM Assembly Language Tools User's Guide* provides a complete description of the linker.
- The **archiver** allows you to collect a group of files into a single archive file, called a *library*. Additionally, the archiver allows you to modify a library by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object files. The *ARM Assembly Language Tools User's Guide* explains how to use the archiver.
- The **run-time-support libraries** contain the standard ISO C and C++ library functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the compiler. See Chapter 7.

  You can use the **library-build utility** to build your own customized run-time-support library. See Section 7.4. Source code for the standard run-time-support library functions for C and C++ are provided in the self-contained rtssrc.zip file.
- The **hex conversion utility** converts an object file into other object formats. You can download the converted file to an EPROM programmer. The *ARM Assembly Language Tools User's Guide* explains how to use the hex conversion utility and describes all supported formats.
- The **absolute lister** accepts linked object files as input and creates .abs files as output. You can assemble these .abs files to produce a listing that contains absolute, rather than relative, addresses. Without the absolute lister, producing such a listing would be tedious and would require many manual operations. The *ARM Assembly Language Tools User's Guide* explains how to use the absolute lister.
- The **cross-reference lister** uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files. The *ARM Assembly Language Tools User's Guide* explains how to use the cross-reference utility.
- The **C++ name demangler** is a debugging aid that converts names mangled by the compiler back to their original names as declared in the C++ source code. As shown in Figure 1-1, you can use the C++ name demangler on the assembly file that is output by the compiler; you can also use this utility on the assembler listing file and the linker map file. See Chapter 8.
- The **stack depth profiler** provides information about the maximum stack depth requirements of an application based on the static information available in the linker output file. See Chapter 9.
- The **disassembler** decodes object files to show the assembly instructions that they represent. The *ARM Assembly Language Tools User's Guide* explains how to use the disassembler.
- The main product of this development process is an executable object file that can be executed in a **ARM** device.

1.2 C/C++ Compiler Overview

The following subsections describe the key features of the compiler.

1.2.1 ANSI/ISO Standard

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

- **ISO-standard C**

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

  The C/C++ compiler conforms to the C++ Standard ISO/IEC 14882:1998. The language is also

Copyright © 2012, Texas Instruments Incorporated

described in Ellis and Stroustrup's *The Annotated C++ Reference Manual* (ARM), but this is not the standard. The compiler also supports embedded C++. For a description of *unsupported* C++ features, see Section 5.2.

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

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

1.2.2 Output Files

These types of output files are created by the compiler:

- **COFF object files**

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

- **ELF object files**

  Executable and linking format (ELF) enables supporting modern language features like early template instantiation and exporting inline functions.

1.2.3 Compiler Interface

These features enable interfacing with the compiler:

- **Compiler program**

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

- **Flexible assembly language interface**

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

1.2.4 Utilities

These features are compiler utilities:

- **Library-build utility**

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

- **C++ name demangler**

  The C++ name demangler (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.

- **Static stack depth profiler**

  The static stack depth profiler provides information about the maximum stack depth requirements of an application based on the static information available in the linker output file. The application must be compiled with DWARF (default) type debug information.

- **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 COFFor 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*.

Copyright © 2012, Texas Instruments Incorporated

Chapter 2
SPNU151H–August 2012

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 object file. All of these steps are executed at once by using the compiler.

Topic **Page**

2.1 About the Compiler 17
2.2 Invoking the C/C++ Compiler 17
2.3 Changing the Compiler's Behavior With Options 18
2.4 Controlling the Compiler Through Environment Variables 35
2.5 Precompiled Header Support 37
2.6 Controlling the Preprocessor 38
2.7 Understanding Diagnostic Messages 41
2.8 Other Messages 44
2.9 Generating Cross-Reference Listing Information (--gen_acp_xref Option) 44
2.10 Generating a Raw Listing File (--gen_acp_raw Option) 45
2.11 Using Inline Function Expansion 46
2.12 Using Interlist 47
2.13 Controlling Application Binary Interface 48
2.14 VFP Support 49
2.15 Enabling Entry Hook and Exit Hook Functions 50

Copyright © 2012, Texas Instruments Incorporated

2.1 About the Compiler

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

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

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

Invoking the Linker

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

For a complete description of the assembler and the linker, see the *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 files, assembly language source files, linear assembly files, or object files.
--run_linker	Option that invokes the linker. The --run_linker option's short form is -z. See Chapter 4 for more information.
link_options	Options that control the linking process.
object files	Name of the additional object files for the linking process.

The arguments to the compiler are of three types:

- Compiler options
- Link options
- Filenames

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

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

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

```
armcl symtab.c file.c seek.asm --run_linker --library=lnk.cmd
       --library=rtsv4_A_be_eabi.lib --output_file=myprogram.out
```


Copyright © 2012, Texas Instruments Incorporated

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:

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

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

Table 2-7 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 }	-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), or ARM V7R (Cortex-R4). The default is ARM V4.	Section 2.3.4
--code_state={ 16 | 32 }	-mt	Designates the ARM compilation mode	Section 2.3.4
--float_support={vfpv3 | vfpv3d16 | fpalib |vfplib | fpv4spd16 }		Generates vector floating-point (VFP) coprocessor instructions	Section 2.14
--abi={ eabi | ti_arm9_abi | tiabi }	-abi	Specifies the application binary interface. The default is --abi=eabi.	Section 2.13
--little_endian	-me	Designates little-endian code	Section 2.3.4

Table 2-2. Optimization Options[(1)]

Option	Alias	Effect	Section
--opt_level=0	-O0	Optimizes register usage	Section 3.1
--opt_level=1	-O1	Uses -O0 optimizations and optimizes locally	Section 3.1
--opt_level=2	-O2 or -O	Uses -O1 optimizations and optimizes globally	Section 3.1
--opt_level=3	-O3	Uses -O2 optimizations and optimizes the file (default)	Section 3.1 Section 3.2
--opt_level=4	-O4	Uses -O3 optimizations and performs link-time optimization	Section 3.4
--opt_for_cache		Works with --opt_for_speed to better optimize instruction caching.	Section 3.10
--opt_for_speed[=*n*]	-mf	Controls speed over space (0-5 range) (Default is 4.)	Section 3.10
--fp_mode={relaxed|strict}		Enables or disables relaxed floating-point mode	Section 2.3.3

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

Copyright © 2012, Texas Instruments Incorporated

Table 2-3. Advanced Optimization Options[(1)]

Option	Alias	Effect	Section
-ab=*num*		Controls the depth of branch chaining optimization	Section 3.11.18
--auto_inline=[*size*]	-oi	Sets automatic inlining size (--opt_level=3 only). If *size* is not specified, the default is 1.	Section 3.7
--call_assumptions=0	-op0	Specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler	Section 3.3.1
--call_assumptions=1	-op1	Specifies that the module contains variables modified from outside the source code provided to the compiler but does not use functions called from outside the source code	Section 3.3.1
--call_assumptions=2	-op2	Specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler (default)	Section 3.3.1
--call_assumptions=3	-op3	Specifies that the module contains functions that are called from outside the source code provided to the compiler but does not use variables modified from outside the source code	Section 3.3.1
--gen_opt_info=0	-on0	Disables the optimization information file	Section 3.2.2
--gen_opt_info=1	-on1	Produces an optimization information file	Section 3.2.2
--gen_opt_info=2	-on2	Produces a verbose optimization information file	Section 3.2.2
--optimizer_interlist	-os	Interlists optimizer comments with assembly statements	Section 3.8
--remove_hooks_when_inlining		Removes entry/exit hooks for auto-inlined functions	Section 2.15
--single_inline		Inlines functions that are only called once	--
--aliased_variables	-ma	Indicates that a specific aliasing technique is used	Section 3.5

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

Table 2-4. Debug Options

Option	Alias	Effect	Section
--symdebug:dwarf	-g	Enables symbolic debugging	Section 2.3.5 Section 3.9.1
--symdebug:coff		Enables symbolic debugging using the alternate STABS debugging format. STABS format is not supported for ELF.	Section 2.3.5 Section 3.9.1
--symdebug:none		Disables all symbolic debugging	Section 2.3.5
--symdebug:profile_coff		Enables profiling using the alternate STABS debugging format. STABS format is not supported for ELF.	Section 2.3.5
--symdebug:skeletal		Enables minimal symbolic debugging that does not hinder optimizations (default behavior)	Section 2.3.5
--optimize_with_debug	-mn	Reenables optimizations disabled with --symdebug:dwarf	Section 3.9.1
--symdebug:dwarf_version=2|3		Specifies the DWARF format version	Section 2.3.5

Table 2-5. Advanced Debug Options

Option	Alias	Effect	Section
--symdebug:keep_all_types		Keep unreferenced type information (default for ELF with debug)	Section 2.3.5

Table 2-6. Include Options

Option	Alias	Effect	Section
--include_path=*directory*	`-I`	Defines #include search path	Section 2.6.2.1
--preinclude=*filename*		Includes *filename* at the beginning of compilation	Section 2.3.3

Copyright © 2012, Texas Instruments Incorporated

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
--run_linker	-z	Enables linking	Section 2.3.2
--skip_assembler	-n	Compiles or assembly optimizes only	Section 2.3.2

Table 2-8. Language Options

Option	Alias	Effect	Section
--cpp_default	-fg	Processes all source files with a C extension as C++ source files.	Section 2.3.7
--create_pch=*filename*		Creates a precompiled header file with the name specified	Section 2.5
--embedded_cpp	-pe	Enables embedded C++ mode	Section 5.15.3
--exceptions		Enables C++ exception handling	Section 5.6
--extern_c_can_throw		Allow extern C functions to propagate exceptions (EABI only)	--
--float_operations_allowed={none|all|32|64}		Restricts the types of floating point operations allowed.	Section 2.3.3
--gcc		Enables support for GCC extensions	Section 5.16
--gen_acp_raw	-pl	Generates a raw listing file	Section 2.10
--gen_acp_xref	-px	Generates a cross-reference listing file	Section 2.9
--keep_unneeded_statics		Keeps unreferenced static variables.	Section 2.3.3
--kr_compatible	-pk	Allows K&R compatibility	Section 5.15.1
--multibyte_chars	-pc	Enables support for multibyte character sequences in comments, string literals and character constants.	--
--no_inlining	-pi	Disables definition-controlled inlining (but --opt_level=3 (or -O3) optimizations still perform automatic inlining)	Section 2.11
--no_intrinsics	-pn	Disables intrinsic functions. No predefinition of compiler-supplied intrinsic functions.	--
--pch		Creates or uses precompiled header files	Section 2.5
--pch_dir=*directory*		Specifies the path where the precompiled header file resides	Section 2.5.2
--pch_verbose		Displays a message for each precompiled header file that is considered but not used	Section 2.5.3
--program_level_compile	-pm	Combines source files to perform program-level optimization	Section 3.3
--relaxed_ansi	-pr	Enables relaxed mode; ignores strict ISO violations	Section 5.15.2
--rtti	-rtti	Enables run time type information (RTTI)	–-
--static_template_instantiation		Instantiate all template entities with internal linkage	–-
--strict_ansi	-ps	Enables strict ISO mode (for C/C++, not K&R C)	Section 5.15.2
--use_pch=*filename*		Specifies the precompiled header file to use for this compilation	Section 2.5.2

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 2.6.7
--preproc_includes[=*filename*]	-ppi	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive	Section 2.6.8
--preproc_macros[=*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 2.6.9
--preproc_only	-ppo	Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension.	Section 2.6.3

Copyright © 2012, Texas Instruments Incorporated

Table 2-9. Parser Preprocessing Options (continued)

Option	Alias	Effect	Section
--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 2.6.5
--preproc_with_compile	-ppa	Continues compilation after preprocessing	Section 2.6.4
--preproc_with_line	-ppl	Performs preprocessing only. Writes preprocessed output with line-control information (#line directives) to a file with the same name as the input but with a .pp extension.	Section 2.6.6

Table 2-10. Predefined Symbols 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. Diagnostics Options

Option	Alias	Effect	Section
--compiler_revision		Prints out the compiler release revision and exits	--
--diag_error=*num*	-pdse	Categorizes the diagnostic identified by *num* as an error	Section 2.7.1
--diag_remark=*num*	-pdsr	Categorizes the diagnostic identified by *num* as a remark	Section 2.7.1
--diag_suppress=*num*	-pds	Suppresses the diagnostic identified by *num*	Section 2.7.1
--diag_warning=*num*	-pdsw	Categorizes the diagnostic identified by *num* as a warning	Section 2.7.1
--display_error_number	-pden	Displays a diagnostic's identifiers along with its text	Section 2.7.1
--emit_warnings_as_errors	-pdew	Treat warnings as errors	Section 2.7.1
--gen_aux_user_info		Generate user information file (.aux)	--
--issue_remarks	-pdr	Issues remarks (nonserious warnings)	Section 2.7.1
--no_warnings	-pdw	Suppresses warning diagnostics (errors are still issued)	Section 2.7.1
--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 diagnostics that display the original source with line-wrap	Section 2.7.1
--write_diagnostics_file	-pdf	Generates a diagnostics information file. Compiler only option.	Section 2.7.1

Copyright © 2012, Texas Instruments Incorporated

Table 2-12. Run-Time Model Options

Option	Alias	Effect	Section
--align_structs=*bytecount*	-mw	Aligns structures on *bytecount* boundary	Section 2.3.4
--disable_branch_chaining		Disables branch chaining optimization	Section 3.11.18
--disable_dual_state	-md	Disables dual-state interworking support (tiabi only)	Section 6.10.1
--embedded_constants={on|off}		Controls whether compiler embeds constants in functions.	Section 2.3.4
--enum_type={int|packed|unpacked}		Designates the underlying type of an enumeration type; default is unpacked	Section 2.3.4
--fp_reassoc={on|off}		Enables or disables the reassociation of floating-point arithmetic	Section 2.3.4
--gen_func_subsections={on|off}	-ms	Puts each function in a separate subsection in the object file	Section 4.2.2
-ml		Uses far calls (requires -md and --code_state=16)	Section 6.4.5
-mo		Enables dynamic stack overflow checking	Section 2.3.4
-neon		Enables support for the Cortex-A8 Neon SIMD instruction set.	Section 2.3.4
--plain_char={signed|unsigned}	-mc	Specifies how to treat plain chars, default is unsigned	Section 2.3.4
--profile:breakpt		Enables breakpoint-based profiling	Section 2.3.5 Section 3.9.2
--profile:power		Enables power profiling	Section 2.3.5 Section 3.9.2
-rr={r5|r6|r9}		Disallows use of rx=[5,6,9] by the compiler	Section 2.3.4
--sat_reassoc={on|off}		Enables or disables the reassociation of saturating arithmetic. Default is --sat_reassoc=off.	Section 2.3.3
--small_enum		Uses the smallest possible size for the enumeration type	Section 2.3.4
--static_template_instantiation		Instantiates all template entities as needed	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
--vectorize=off		Disables vectorization of source code	Section 2.3.4

Table 2-13. 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

Table 2-14. Library Function Assumptions Options

Option	Alias	Effect	Section
--printf_support={nofloat|full|minimal}		Enables support for smaller, limited versions of the printf and sprintf run-time-support functions.	Section 2.3.3
--std_lib_func_defined	-ol1 or -oL1	Informs the optimizer that your file declares a standard library function	Section 3.2.1
--std_lib_func_not_defined	-ol2 or -oL2	Informs the optimizer that your file does not declare or alter library functions. Overrides the -ol0 and -ol1 options (default).	Section 3.2.1
--std_lib_func_redefined	-ol0 or -oL0	Informs the optimizer that your file alters a standard library function	Section 3.2.1

Table 2-15. Assembler Options

Option	Alias	Effect	Section
--keep_asm	-k	Keeps the assembly language (.asm) file	Section 2.3.11
--asm_listing	-al	Generates an assembly listing file	Section 2.3.11

Copyright © 2012, Texas Instruments Incorporated

Table 2-15. Assembler Options (continued)

Option	Alias	Effect	Section
--c_src_interlist	-ss	Interlists C source and assembly statements	Section 2.12 Section 3.8
--src_interlist	-s	Interlists optimizer comments (if available) and assembly source statements; otherwise interlists C and assembly source statements	Section 2.3.2
--absolute_listing	-aa	Enables absolute listing	Section 2.3.11
--asm_define=*name*[=*def*]	-ad	Sets the *name* symbol	Section 2.3.11
--asm_dependency	-apd	Performs preprocessing; lists only assembly dependencies	Section 2.3.11
--asm_includes	-api	Performs preprocessing; lists only included #include files	Section 2.3.11
--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
--copy_file=*filename*	-ahc	Copies the specified file for the assembly module	Section 2.3.11
--cross_reference	-ax	Generates the cross-reference file	Section 2.3.11
--force_thumb2_mode={true|false}		Controls assembly optimization of 32-bit Thumb2 into 16-bit Thumb instructions	Section 2.3.11
--include_file=*filename*	-ahi	Includes the specified file for the assembly module	Section 2.3.11
--max_branch_chain=*num*		Controls the depth of branch chaining through the assembler	Section 3.11.18
--no_const_clink		Stops generation of .clink directives for const global arrays.	Section 2.3.3
--output_all_syms	-as	Puts labels in the symbol table	Section 2.3.11
--syms_ignore_case	-ac	Makes case insignificant in assembly source files	Section 2.3.11
--ual		Generates UAL syntax	Section 2.3.11

Table 2-16. File Type Specifier Options

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

Table 2-17. Directory Specifier Options

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

Copyright © 2012, Texas Instruments Incorporated

Table 2-18. Default File Extensions Options

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

Table 2-19. Command Files Options

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

Table 2-20. MISRA-C:2004 Options

Option	Alias	Effect	Section
--check_misra[={all|required|advisory|none|*rulespec*}]		Enables checking of the specified MISRA-C:2004 rules. Default is all.	Section 2.3.3
--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 the *ARM Assembly Language Tools User's Guide* for details on these options.

Table 2-21. Linker Basic Options

Option	Alias	Description
--output_file=*file*	-o	Names the executable output file. The default filename is a.out.
--map_file=*file*	-m	Produces a map or listing of the input and output sections, including holes, and places the listing in *filename*
--stack_size=*size*	[-]-stack	Sets 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 2-22. File Search Path Options

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

Copyright © 2012, Texas Instruments Incorporated

Table 2-23. Command File Preprocessing Options

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

Table 2-24. Diagnostic Options

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

Table 2-25. Linker Output Options

Option	Alias	Description
--absolute_exe	-a	Produces an absolute, executable object file. This is the default; if neither --absolute_exe nor --relocatable is specified, the linker acts as if --absolute_exe were specified.
--generate_dead_funcs_list		Writes a list of the dead functions that were removed by the linker to file fname.
--mapfile_contents=*attribute*		Controls the information that appears in the map file.
--relocatable	-r	Produces a nonexecutable, relocatable output object file
--rom		Creates a ROM object
--run_abs	-abs	Produces an absolute listing file
--xml_link_info=*file*		Generates a well-formed XML *file* containing detailed information about the result of a link

Table 2-26. Symbol Management Options

Option	Alias	Description
--entry_point=*symbol*	-e	Defines a global symbol that specifies the primary entry point for the executable object file
--globalize=*pattern*		Changes the symbol linkage to global for symbols that match *pattern*
--hide=*pattern*		Hides symbols that match the specified *pattern*
--localize=*pattern*		Make the symbols that match the specified *pattern* local
--make_global=*symbol*	-g	Makes *symbol* global (overrides -h)
--make_static	-h	Makes all global symbols static
--no_sym_merge	-b	Disables merge of symbolic debugging information in COFF object files
--no_symtable	-s	Strips symbol table information and line number entries from the executable object file
--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

Copyright © 2012, Texas Instruments Incorporated

Table 2-26. Symbol Management Options (continued)

Option	Alias	Description
--undef_sym=*symbol*	-u	Adds *symbol* to the symbol table as an unresolved symbol
--unhide=*pattern*		Excludes symbols that match the specified *pattern* from being hidden

Table 2-27. Run-Time Environment Options

Option	Alias	Description
--arg_size=*size*	--args	Reserve *size* bytes for the argc/argv memory area
-be32		Forces the linker to generate BE-32 object code.
-be8		Forces the linker to generate BE-8 object code.
--fill_value=*value*	-f	Sets default fill value for holes within output sections
--ram_model	-cr	Initializes variables at load time
--rom_model	-c	Autoinitializes variables at run time
--trampolines[=off|on]		Generates far call trampolines (argument is optional, is "on" by default)

Table 2-28. Link-Time Optimization Options

Option	Description
--cinit_compression[=compression_kind]	Specifies the type of compression to apply to the c auto initialization data. Default is rle.
--compress_dwarf[=off|on]	Aggressively reduces the size of DWARF information from input object files. Default is on.
--copy_compression[=compression_kind]	Compresses data copied by linker copy tables. Default is rle.
--unused_section_elimination[=off|on]	Eliminates sections that are not needed in the executable module. Default is on.

Table 2-29. Miscellaneous Options

Option	Alias	Description
--disable_clink	-j	Disables conditional linking of COFF object files
--linker_help	[-]-help	Displays information about syntax and available options
--minimize_trampolines		Places sections to minimize number of far trampolines required
--preferred_order=*function*		Prioritizes placement of functions
--strict_compatibility[=off|on]		Performs more conservative and rigorous compatibility checking of input object files. Default is on.
--trampoline_min_spacing		When trampoline reservations are spaced more closely than the specified limit, tries to make them adjacent
--zero_init=[off|on]		Controls preinitialization of uninitialized variables. Default is on.

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 3.8. The --c_src_interlist option can have a negative performance and/or code size impact.

Copyright © 2012, Texas Instruments Incorporated

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

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

```
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, the *name* is set to 1. The --define option's short form is -D.

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

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

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

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

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

--keep_asm Retains the assembly language output from the compiler or assembly optimizer. Normally, the compiler deletes the output assembly language file after assembly is complete. The --keep_asm option's short form is -k.

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

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

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

Copyright © 2012, Texas Instruments Incorporated

--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:

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

--float_operations_allowed={none|all|32|64} Restricts the type of floating point operations allowed in the application. The default is all. If set to none, 32, or 64, the application is checked for operations that will be performed at runtime. For example, declared variables that are not used will not cause a diagnostic. The checks are performed after relaxed mode optimizations have been performed, so illegal operations that are completely removed result in no diagnostics.

--fp_mode={relaxed|strict} The default floating-point mode is strict. Relaxed floating-point mode causes double-precision floating-point computations and storage to be converted to single-precision floating-point or integers 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 the result of a double-precision floating-point expression is assigned to a single-precision floating-point or an integer or immediately used in a single-precision context, the computations in the expression are converted to single-precision computations. Double-precision constants in the expression are also 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.

--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.

Copyright © 2012, Texas Instruments Incorporated

--keep_unneeded_statics	Does not delete unreferenced static variables. The parser by default remarks about and then removes any unreferenced static variables. The --keep_unneeded_statics option keeps the parser from deleting unreferenced static variables and any static functions that are referenced by these variable definitions. Unreferenced static functions will still be removed.
--no_const_clink	Tells the compiler to not generate .clink directives for const global arrays. By default, these arrays are placed in a .const subsection and conditionally linked.
--misra_advisory={error|warning|remark|suppress}	Sets the diagnostic severity for advisory MISRA-C:2004 rules.
--misra_required={error|warning|remark|suppress}	Sets the diagnostic severity for required MISRA-C:2004 rules.
--preinclude=*filename*	Includes the source code of *filename* at the beginning of the compilation. This can be used to establish standard macro definitions. The filename is searched for in the directories on the include search list. The files are processed in the order in which they were specified.
--printf_support={full|nofloat|minimal}	Enables support for smaller, limited versions of the printf and sprintf run-time-support functions. The valid values are: • full: Supports all format specifiers. This is the default. • nofloat: Excludes support for printing and scanning floating-point values. Supports all format specifiers except %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.

-ab=*num*	Controls the depth of branch chaining. See Section 3.11.18.
--abi={eabi|ti_arm9_abi|tiabi}	Specifies the application binary interface (ABI). The default is --abi=eabi, which specifies the ABI defined by the industry consortium founded by ARM Limited. With the --abi=ti_arm9_abi setting, the linker generates either a BLX instruction or a veneer if there is a mode change. Therefore, no label prefixes ($ for 16-BIS or _ for 32-BIS) are generated. Thus, new object files do not link with older object files. The --abi=tiabi option has been deprecated; it is recommended that those still using this ABI move to TI_ARM9_ABI. The --abi=tiabi option preserves the old behavior (prefixes) for compatibility with object files produced prior to version 4.1.0. See Section 2.13 .

Copyright © 2012, Texas Instruments Incorporated

--align_structs=*bytecount* Forces alignment of structures to a minimum *bytecount*-byte boundary, where bytecount is a power of 2. To align all structs to a word boundary use --align_structs=4. All structs in the file will contain the bytecount minimum alignment, including nested structs. Only a minimum alignment is set, the data structures are not packed. Your program may break if one file is compiled with --align_structs and another is not, or is a different alignment is used. The offsets of a nested switch could be incorrect if different alignments are used.

--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 (or -mt) option generates Thumb-2 code. For details on indirect calls in 16-bit versus 32-bit code, see Section 6.10.2.2.

--disable_branch_chaining Disables the branch chaining optimization. See Section 3.11.18.

--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|unpacked} Designates the underlying type of an enumeration type. The default is unpacked, which designates the underlying type as int if the enumerator constants can be represented in int. In C++, the underlying type is long long if an enumerator constant cannot be represented by int or unsigned int.

Using --enum_type=packed forces the enumeration type to be packed, which designates the underlying type of the enumeration is chosen to be the smallest integer that accommodates the enumerator constants.

Using --enum_type=int designates the underlying type to always int. An enumerator with a value outside int range generates an error.

--float_support={ vfpv3 | vfpv3d16 | fpalib | vfplib | fpv4spd16 } Generates vector floating-point (VFP) coprocessor instructions for various versions and libraries. See Section 2.14.

-md Disables dual-state interworking support. See Section 6.10.1.

-ml Use long calls (requires -md and --code_state). See Section 6.4.5.

-mo Enables dynamic stack overflow checking.

-mv={4|5e|6|6M0|7A8|7M3|7M4|7R4} 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), or ARM V7R (Cortex-R4). 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.

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

-r={r5|r6|r9} Disallows use of rx=[5|6|9] by the compiler.

Copyright © 2012, Texas Instruments Incorporated

--small_enum — By default, the ARM compiler uses 32 bits for every enum. When you use the --small_enum option, the smallest possible byte size for the enumeration type is used. For example, enum example_enum {first = -128, second = 0, third = 127} uses only one byte instead of 32 bits when the --small_enum option is used. Similarly, enum a_short_enum {bottom = -32768, middle = 0, top = 32767} fits into two bytes instead of four. Do not link object files compiled with the --small_enum option with object files that have been compiled without it. If you use the --small_enum option, you must use it with all of your C/C++ files; otherwise, you will encounter errors that cannot be detected until run time.

--static_template_instantiation — Instantiates all template entities in the current file as needed though the parser. These instantiations are also given internal (static) linkage.

--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.

--vectorize=**off** — Disables the vectorization of source code. The optimizer automatically vectorizes code when -mv7A8 --neon is specified.

2.3.5 Symbolic Debugging and Profiling Options

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

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

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

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

--symdebug:dwarf — Generates directives that are used by the C/C++ source-level debugger and enables assembly source debugging in the assembler. The --symdebug:dwarf option's short form is -g. The --symdebug:dwarf option disables many code generator optimizations, because they disrupt the debugger. You can use the --symdebug:dwarf option with the --opt_level (aliased as -O) option to maximize the amount of optimization that is compatible with debugging (see Section 3.9.1).

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

Copyright © 2012, Texas Instruments Incorporated

--symdebug:dwarf_version={2|3} Specifies the DWARF debugging format version (2 or 3) to be generated when --symdebug:dwarf or --symdebug:skeletal is specified. By default, the compiler generates DWARF version 3 debug information. For more information on TI extensions to the DWARF language, see *The Impact of DWARF on TI Object Files* (SPRAAB5).

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

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

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

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

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

2.3.6 Specifying Filenames

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

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

NOTE: Case Sensitivity in Filename Extensions

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

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

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

```
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.

Copyright © 2012, Texas Instruments Incorporated

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.

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
```


Copyright © 2012, Texas Instruments Incorporated

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

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

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

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

--asm_listing Produces an assembly listing file.

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

--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 (or -mt) option generates Thumb-2 code. For details on indirect calls in 16-bit versus 32-bit code, see Section 6.10.2.2.

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

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

Copyright © 2012, Texas Instruments Incorporated

--force_thumb2_mode={true|false} — Alters default assembler behavior. By default, for C or C++ code, the assembler optimizes 32-bit Thumb2 instructions when possible. For hand-coded assembly code, the assembler does not optimize 32-bit Thumb2 instructions.

If you require no assembly optimization of 32-bit instructions for C or C++, use --force_thumb2_mode=true. If you require assembly optimization of 32-bit instructions in assembly files, use --force_thumb2_mode=false.

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

--max_branch_chain — Controls the depth of branch chaining through the assembler. See Section 3.11.18.

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

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

--ual — Accepts UAL syntax when assembling for ARMv6 and earlier architectures. UAL syntax is the default for ARMv7 and later architectures.

2.3.12 Deprecated Options

Several compiler options have been deprecated. The compiler continues to accept these options, but they are not recommended for use. Future releases of the tools will not support these options. Table 2-30 lists the deprecated options and the options that have replaced them.

Table 2-30. Compiler Backwards-Compatibility Options Summary

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

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

2.4 Controlling the Compiler Through Environment Variables

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

NOTE: C_OPTION and C_DIR

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

2.4.1 Setting Default Compiler Options (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.

Copyright © 2012, Texas Instruments Incorporated

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.

In the following examples, each time you run the compiler, it runs the linker. Any options following --run_linker on the command line or in 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 an Alternate Directory (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:

Copyright © 2012, Texas Instruments Incorporated

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 Precompiled Header Support

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

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

2.5.1 Automatic Precompiled Header

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

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

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

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

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

2.5.2 Manual Precompiled Header

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

--create_pch=*filename*

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

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

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

2.5.3 Additional Precompiled Header Options

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

Copyright © 2012, Texas Instruments Incorporated

2.6 Controlling the Preprocessor

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

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

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

2.6.1 Predefined Macro Names

The compiler maintains and recognizes the predefined macro names listed in Table 2-31.

Table 2-31. Predefined ARM Macro Names

Macro Name	Description
_ _16bis_ _	Defined if 16-BIS state is selected (the -mt option is used); otherwise, it is undefined.
_ _32bis_ _	Defined if 32-BIS state is selected (the -mt 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
_ _LINE_ _[(1)]	Expands to the current line number
_ _signed_chars_ _	Defined if char types are signed by default
_ _STDC_ _[(1)]	Defined to indicate that compiler conforms to ISO C Standard. See Section 5.1 for exceptions to ISO C conformance.
_ _STDC_VERSION_ _	C standard macro
_ _TI_COMPILER_VERSION_ _	Defined to a 7-9 digit integer, depending on if X has 1, 2, or 3 digits. The number does not contain a decimal. For example, version 3.2.1 is represented as 3002001. The leading zeros are dropped to prevent the number being interpreted as an octal.
_ _TI_EABI_SUPPORT_ _	Defined to 1 if the EABI ABI is enabled (the --abi=eabi option is used; 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 if GCC extensions are enabled (the --gcc option is used); otherwise, it is undefined.
_ _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 if strict ANSI/ISO mode is enabled (the --strict_ansi option is used); otherwise, it is undefined.
_ _TI_STRICT_FP_MODE_ _	Defined to 1 if --fp_mode=strict is used (or implied); otherwise, it is undefined.
_ _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.

(1) Specified by the ISO standard

Copyright © 2012, Texas Instruments Incorporated

Table 2-31. Predefined ARM Macro Names (continued)

Macro Name	Description
_ _TI_ ARM_V7R4_ _	Defined to 1 if the v7R4 architecture (Cortex-R4) is targeted (the -mv7R4 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.
_ _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_FPV4SPD16_SUPPORT_ _	Defined to 1 if the VFP coprocessor is enabled (the --float_support=fpv4spd16 option is used); otherwise, it is undefined.
_ _TIME_ _ [(1)]	Expands to the compilation time in the form "*hh:mm:ss*"
_ _TI_ ARM_ _	Always defined
_ _unsigned_chars_ _	Defined if char types are unsigned by default (default)
_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.
_INLINE	Expands to 1 if optimization is used (--opt_level or -O option); undefined otherwise. Regardless of any optimization, always undefined when --no_inlining is used.
_little_endian_ _	Defined if little-endian mode is selected (the --endian=little option is used); otherwise, it is undefined.

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 2-31 in the same manner as any other defined name. For example,

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

translates to a line such as:

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

2.6.2 The Search Path for #include Files

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

- If you enclose the filename in double quotes (" "), the compiler searches for the file in the following directories in this order:
  1. The directory of the file that contains the #include directive and in the directories of any files that contain that file.
  2. Directories named with the --include_path option.
  3. Directories set with the 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 2.6.2.1 for information on using the --include_path option. See Section 2.4.2 for more information on input file directories.

Copyright © 2012, Texas Instruments Incorporated

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

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

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

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

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

Assume that the complete pathname for alt.h is:

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

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

Operating System	Enter
UNIX	`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
```

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

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

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

Copyright © 2012, Texas Instruments Incorporated

2.6.4 Continuing Compilation After Preprocessing (--preproc_with_compile Option)

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

2.6.5 Generating a Preprocessed Listing File With Comments (--preproc_with_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.

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

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

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

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

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

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

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

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

2.7 Understanding Diagnostic Messages

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

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

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

Copyright © 2012, Texas Instruments Incorporated

Diagnostic messages have an associated severity, as follows:

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

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

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

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

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

Long messages are wrapped to additional lines, when necessary.

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

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

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

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

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

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

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

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

Copyright © 2012, Texas Instruments Incorporated

2.7.1 Controlling Diagnostics

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

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

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

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

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

--display_error_number Displays a diagnostic's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (--diag_suppress, --diag_error, --diag_remark, and --diag_warning). This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix -D; otherwise, no suffix is present. See Section 2.7.

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

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

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

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

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

2.7.2 How You Can Use Diagnostic Suppression Options

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

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


Copyright © 2012, Texas Instruments Incorporated

```
    break;
    }
}
```

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

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

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

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

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

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

2.8 Other Messages

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

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

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

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

sym-id name X filename line number column number

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

Copyright © 2012, Texas Instruments Incorporated

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

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

The raw listing file contains the following information:

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

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

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_acp_raw 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

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

Copyright © 2012, Texas Instruments Incorporated

2.11 Using Inline Function Expansion

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

Inline function expansion is performed in one of the following ways:

- Intrinsic operators are inlined by default.
- Code is compiled with definition-controlled inlining.
- When the optimizer is invoked with the --opt_level=3 option (-O3) (the default), automatic inline expansion is performed at call sites to small functions. For more information about automatic inline function expansion, see Section 3.7.

NOTE: Function Inlining Can Greatly Increase Code Size

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

2.11.1 Inlining Intrinsic Operators

An operator is intrinsic if it can be implemented very efficiently with the target's instruction set. The compiler automatically inlines the intrinsic operators of the target system by default. Inlining happens whether or not you use the optimizer and whether or not you use any compiler or optimizer options on the command line. These functions are considered the intrinsic operators:

- abs
- labs
- fabs

2.11.2 Using the inline Keyword, the --no_inlining Option, and Level 3 Optimization

Definition-controlled inline function expansion is performed when you invoke the compiler with optimization and the compiler encounters the inline keyword in code. Functions with a variable number of arguments are not inlined. In addition, a limit is placed on the depth of inlining for recursive or nonleaf functions. Inlining should be used for small functions or functions that are called in a few places (though the compiler does not enforce this). You can control this type of function inlining with the inline keyword.

The inline keyword specifies that a function is expanded inline at the point at which it is called, rather than by using standard calling procedures.

The semantics of the inline keyword follows that described in the C++ standard. The inline keyword is identically supported in C as a language extension. Because it is a language extension that could conflict with a strictly conforming program, however, the keyword is disabled in strict ANSI C mode (when you use the --strict_ansi compiler option). If you want to use definition-controlled inlining while in strict ANSI C mode, use the alternate keyword _ _inline.

When you want to compile without definition-controlled inlining, use the --no_inlining option.

NOTE: Using the --no_inlining Option With Level 3 Optimizations

When you use the --no_inlining option with --opt_level=3 (the default, aliased as -O3) optimizations, automatic inlining is still performed.

2.11.3 Automatic Inlining

When optimizing with the --opt_level=3 or --opt_level=2 option (aliased as -O3 or -O2), the compiler automatically inlines certain functions. For more information, see Section 3.7.

Copyright © 2012, Texas Instruments Incorporated

2.11.4 Inlining Restrictions

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

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

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

2.12 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.

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

Example 2-1 shows a typical interlisted assembly file.

For more information about using the interlist feature with the optimizer, see Section 3.8.

Example 2-1. An Interlisted Assembly Language File

```
_main:
        STMFD      SP!, {LR}
;------------------------------------------------------------------------------
;    5 | printf("Hello, world\n");
;------------------------------------------------------------------------------
        ADR        A1, SL1
        BL         _printf
;------------------------------------------------------------------------------
```


Copyright © 2012, Texas Instruments Incorporated

Example 2-1. An Interlisted Assembly Language File (continued)

```
;    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 code to link together, regardless of its 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 supports three ABIs. The ABI is chosen through the --abi option as follows:

- **ARM ABIv2** or **EABI** (--abi=eabi)

  The EABI is the default ABI for the ARM compiler. If no --abi option is specified, the compiler uses the EABI. 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.

- **TI ARM9 ABI** (--abi=ti_arm9_abi)

  The TI_ARM9_ABI object format, procedure calling conventions, and all the other details of this ABI are documented throughout this document and in the *ARM Assembly Language Tools User's Guide*.

- **TIABI** (--abi=tiabi)

  The TIABI is an obsolete ABI and is provided only for backward compatibility. If you are using this ABI, we recommend that your move to TI_ARM9_ABI. You must enable the TIABI if you are linking object code or object libraries that have been compiled with an older compiler (ARM 2.x compilers). You must also enable the TIABI if you are compiling hand-coded assembly source files that were initially written for a ARM 2.x compiler. The TIABI mode is only supported on the ARM7 (v4) architecture.

For more details on the different ABIs, see Section 5.11.

Copyright © 2012, Texas Instruments Incorporated

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:

vfpv3— VFPv3 architecture and instruction set

vfpv3d16— VFPv3d16 architecture and instruction set

fpv4spd16 — FPv4-SP architecture and instruction set

fpalib— FPA endianness is used to store double-precision floating-point values (most significant word occupies the lower memory address).

vfplib— VFP endianness is used to store double-precision floating-point values (endianness used is that of the memory system). All VFP coprocessors use this endianness to represent doubles.

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 7.1.7 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_COPROC_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.

Copyright © 2012, Texas Instruments Incorporated

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 5.9.14 for information about the NO_HOOKS pragma.

Copyright © 2012, Texas Instruments Incorporated

Chapter 3

SPNU151H–August 2012

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.

Topic **Page**

3.1 Invoking Optimization 52
3.2 Performing File-Level Optimization (--opt_level=3 option) 53
3.3 Performing Program-Level Optimization (--program_level_compile and --opt_level=3 options) 54
3.4 Link-Time Optimization (--opt_level=4 Option) 56
3.5 Accessing Aliased Variables in Optimized Code 57
3.6 Use Caution With asm Statements in Optimized Code 57
3.7 Automatic Inline Expansion (--auto_inline Option) 57
3.8 Using the Interlist Feature With Optimization 58
3.9 Debugging and Profiling Optimized Code 60
3.10 Controlling Code Size Versus Speed 61
3.11 What Kind of Optimization Is Being Performed? 62

Copyright © 2012, Texas Instruments Incorporated

3.1 Invoking Optimization

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

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

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

  The compiler uses --opt_level=3 (-O3) as the default if you do not use the --opt_level (-O) option at all. You can use --opt_level=off to disable optimization. If debugging is enabled, the compiler defaults to --opt_level=off. If an optimization level of 2 or higher is used, the compiler defaults to --optimize_with_debug=on. You can use --optimize_with_debug=off to disable this behavior.
- **--opt_level=1** or **-O1**

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

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

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

    If you use --opt_level=3 (-O3), see Section 3.2 and Section 3.3 for more information.
- **--opt_level=4** or **-O4**

  Performs link-time optimization. See Section 3.4 for details.

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

Copyright © 2012, Texas Instruments Incorporated

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

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

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

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

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

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

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

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

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

When you invoke the compiler with the --opt_level=3 option (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 3-3 to select the appropriate level to append to the option.

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

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

Copyright © 2012, Texas Instruments Incorporated

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

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

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

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

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

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

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

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

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

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

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

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

Copyright © 2012, Texas Instruments Incorporated

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

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

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

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

If you have any assembly functions in your program, you need to exercise caution when using the --program_level_compile option. The compiler recognizes only the C/C++ source code and not any assembly code that might be present. Because the compiler does not recognize the assembly code calls and variable modifications to C/C++ functions, the --program_level_compile option optimizes out those C/C++ functions. To keep these functions, place the FUNC_EXT_CALLED pragma (see Section 5.9.9) before any declaration or reference to a function that you want to keep.

Another approach you can take when you use assembly functions in your program is to use the --call_assumptions=*n* option with the --program_level_compile and --opt_level=3 options (see Section 3.3.1).

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

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

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

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

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

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

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

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

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

Copyright © 2012, Texas Instruments Incorporated

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

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

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

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

3.4 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 by using the --opt_level=4 option. This option must be used in both the compilation and linking steps. 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.

Link-time optimization provides the same optimization opportunities as program level optimization (Section 3.3), with 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.

3.4.1 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 effect 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.

Copyright © 2012, Texas Instruments Incorporated

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

3.4.2 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.

3.5 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.

3.6 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.

3.7 Automatic Inline Expansion (--auto_inline Option)

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

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

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

Copyright © 2012, Texas Instruments Incorporated

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

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

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

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

Some Functions Cannot Be Inlined

NOTE: For a call-site to be considered for inlining, it must be legal to inline the function and inlining must not be disabled in some way. See the inlining restrictions in Section 2.11.4.

Optimization Level 3 or 2 and Inlining

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

Inlining and Code Size

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

3.8 Using the Interlist Feature With Optimization

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

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

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

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

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

Impact on Performance and Code Size

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

Copyright © 2012, Texas Instruments Incorporated

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

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

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

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

Example 3-2. The Function From Example 2-1 Compiled with the --opt_level=2, --optimizer_interlist, and --c_src_interlist Options

```
_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}
```


Copyright © 2012, Texas Instruments Incorporated

3.9 Debugging and Profiling Optimized Code

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

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

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

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

If debugging is enabled, the compiler defaults to --opt_level=off. If an optimization level of 2 or higher is used, the compiler defaults to --optimize_with_debug=on. You can use --optimize_with_debug=off to disable this behavior.

Symbolic Debugging Options Affect Performance and Code Size

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

3.9.2 Profiling Optimized Code

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

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

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

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

Profile Points

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

Copyright © 2012, Texas Instruments Incorporated

3.10 Controlling Code Size Versus Speed

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

- --opt_for_speed=0

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

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

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

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

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

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

If you specify the option without a parameter, the default setting is --opt_for_speed=4.

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

Copyright © 2012, Texas Instruments Incorporated

3.11 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.

Following are some of the optimizations performed by the compiler:

Optimization	See
Cost-based register allocation	Section 3.11.1
Alias disambiguation	Section 3.11.1
Branch optimizations and control-flow simplification	Section 3.11.3
Data flow optimizations • Copy propagation • Common subexpression elimination • Redundant assignment elimination	Section 3.11.4
Expression simplification	Section 3.11.5
Inline expansion of functions	Section 3.11.6
Function Symbol Aliasing	Section 3.11.7
Induction variable optimizations and strength reduction	Section 3.11.8
Loop-invariant code motion	Section 3.11.9
Loop rotation	Section 3.11.10
Instruction scheduling	Section 3.11.11

ARM-Specific Optimization	See
Tail merging	Section 3.11.12
Autoincrement addressing	Section 3.11.13
Block conditionalizing	Section 3.11.14
Epilog inlining	Section 3.11.15
Removing comparisons to zero	Section 3.11.16
Integer division with constant divisor	Section 3.11.17
Branch chaining	Section 3.11.18

3.11.1 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.

3.11.2 Alias Disambiguation

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

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

Copyright © 2012, Texas Instruments Incorporated

3.11.3 Branch Optimizations and Control-Flow Simplification

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

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

3.11.4 Data Flow Optimizations

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

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

3.11.5 Expression Simplification

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

3.11.6 Inline Expansion of Functions

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

3.11.7 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.

Copyright © 2012, Texas Instruments Incorporated

3.11.8 Induction Variables and Strength Reduction

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

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

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

3.11.9 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.

3.11.10 Loop Rotation

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

3.11.11 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.

3.11.12 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.

3.11.13 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)...
```


Copyright © 2012, Texas Instruments Incorporated

3.11.14 Block Conditionalizing

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

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

Example 3-3. Block Conditionalizing C Source

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

   return ++a;
}
```

Example 3-4. C/C++ Compiler Output for Example 3-3

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

3.11.15 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.

3.11.16 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.

3.11.17 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.

Copyright © 2012, Texas Instruments Incorporated

3.11.18 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 use the compiler --ab=*num* option to control the depth of branch chaining. The depth is determined by the value of *num*. A value of zero tells the compiler not to perform branch chaining. A negative value results in an assembler warning and the branch chaining depth defaults to zero. Alternatively, you can use the assembler --max_branch_chain=*num* option.

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.

Branch chaining is disabled by the --disable_branch_chaining option.

Copyright © 2012, Texas Instruments Incorporated

Linking C/C++ Code

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

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

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

Topic **Page**

4.1 Invoking the Linker Through the Compiler (-z Option) 68
4.2 Linker Code Optimizations 70
4.3 Controlling the Linking Process 71

Copyright © 2012, Texas Instruments Incorporated

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, you must use **--rom_model** or **--ram_model**. The --rom_model option uses automatic variable initialization at run time; the --ram_model option uses variable initialization at load time.
filenames	Names of object files, linker command files, or archive libraries. The default extension for all input files is *.obj*; any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is *a.out*, unless you use the --output_file option to name the output file.
options	Options affect how the linker handles your object files. Linker options can only appear after the **--run_linker** option on the command line, but otherwise may be in any order. (Options are discussed in detail in the *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.

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.obj, prog2.obj, and prog3.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
```


Copyright © 2012, Texas Instruments Incorporated

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
```

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.

Copyright © 2012, Texas Instruments Incorporated

4.2 Linker Code Optimizations

These options are used to further optimize your code.

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

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

--generate_dead_funcs_list= *filename*

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

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

1. Compile all source files using the --gen_func_subsections compiler option. For example:
   ```
   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.obj file2.obj --generate_dead_funcs_list=feedback.txt
   ```
   Alternatively, you can combine steps 1 and 2 into one step. When you do this, you are not required to specify --gen_func_subsections when compiling the source files as this is done for you automatically. For example:
   ```
   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.obj file2.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 --gen_dead_funcs_list is tightly controlled. It must be generated by the linker in order to be processed correctly by the compiler. The format of this file may change over time, so the file contains a version format number to allow backward compatibility.

4.2.2 Generating Function Subsections (--gen_func_subsections Compiler Option)

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

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

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

Copyright © 2012, Texas Instruments Incorporated

In addition to placing each function in a separate subsection, the compiler also annotates that subsection with a conditional linking directive, .clink. This directive marks the section as a candidate to be removed if it is not referenced by any other section in the program. The compiler does not place a .clink directive in a subsection for a trap or interrupt function, as these may be needed by a program even though there is no symbolic reference to them anywhere in the program.

If a section that has been marked for conditional linking is never referenced by any other section in the program, that section is removed from the program. Conditional linking is disabled when performing a partial link or when relocation information is kept with the output of the link. Conditional linking can also be disabled with the --disable_clink link option.

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

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

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

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

Example 4-1. Using the --issue_remarks Option

```
armcl --abi=eabi --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"
```


Copyright © 2012, Texas Instruments Incorporated

4.3.1.2 Manual Run-Time-Support Library Selection

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

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 and secondary system 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 function main
8. Call exit when main returns

A sample bootstrap routine is _c_int00, provided in boot.obj in the run-time support object libraries. The entry point is usually set to the starting address of the bootstrap routine.

NOTE: The _c_int00 Symbol

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

4.3.3 Global Object Constructors

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

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

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

Copyright © 2012, Texas Instruments Incorporated

Section 6.9.2.3 discusses the format of the global constructor table for TI ARM9 ABI and TIABI (deprecated) modes.

Section 6.9.3.6 discusses the format of the global constructor table for EABI mode.

4.3.4 Specifying the Type of Global Variable Initialization

The C/C++ compiler produces data tables for initializing global variables. Section 6.9.4 discusses the format of these initialization tables for TI_ARM9_ABI and TIABI (deprecated) . Section 6.9.3.4 discusses the format of these initialization tables for EABI. The initialization tables are used in one of the following ways:

- Global variables are initialized at *run time*. Use the --rom_model linker option (see Section 6.9.2.1).
- Global variables are initialized at *load time*. Use the --ram_model linker option (see Section 6.9.2.2).

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

When you compile and link programs, the --rom_model option is the default. If used, the --rom_model option must follow the --run_linker option (see Section 4.1). The following list outlines the linking conventions for TI_ARM9_ABI and TIABI used with --rom_model or --ram_model:

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

For details on linking conventions for EABI used with --rom_model and --ram_model, see Section 6.9.3.3 and Section 6.9.3.5, 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.

Copyright © 2012, Texas Instruments Incorporated

4.3.5 Specifying Where to Allocate Sections in Memory

The compiler produces relocatable blocks of code and data. These blocks, called *sections*, are allocated in memory in a variety of ways to conform to a variety of system configurations. See Section 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 created under the TI ARM9 ABI and TIABI (deprecated) modes. Table 4-2 summarizes the initialized sections created under the EABI mode. Table 4-3 summarizes the uninitialized sections. Be aware that the TI ARM9 ABI, TIABI, and EABI .cinit tables have different formats.

Table 4-1. Initialized Sections Created by the Compiler for TI ARM9 ABI and TIABI (deprecated)

Name	Contents
.cinit	Tables for explicitly initialized global and static variables.
.const	Global and static const variables that are explicitly initialized.
.pinit	Table of constructors to be called at startup.
.text	Executable code and constants. Also contains string literals and switch tables. See Section 6.1.1 for exceptions.

Table 4-2. Initialized Sections Created by the Compiler for EABI

Name	Contents
.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.
.text	Executable code and constants. Also contains string literals and switch tables. See Section 6.1.1 for exceptions.

Table 4-3. Uninitialized Sections Created by the Compiler

Name	Contents
.bss	Global and static variables
.stack	Stack
.sysmem	Memory for malloc functions (heap)

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

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*.

Copyright © 2012, Texas Instruments Incorporated

4.3.6 A Sample Linker Command File

Example 4-2 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**

Example 4-2. 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   */
```


Copyright © 2012, Texas Instruments Incorporated

Chapter 5

SPNU151H–August 2012

ARM C/C++ Language Implementation

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

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

Topic **Page**

5.1 Characteristics of ARM C 77
5.2 Characteristics of ARM C++ 77
5.3 Using MISRA-C:2004 78
5.4 Data Types 79
5.5 Keywords 80
5.6 C++ Exception Handling 82
5.7 Register Variables and Parameters 83
5.8 The asm Statement 84
5.9 Pragma Directives 85
5.10 The _Pragma Operator 97
5.11 Application Binary Interface 98
5.12 ARM Instruction Intrinsics 99
5.13 Object File Symbol Naming Conventions (Linknames) 106
5.14 Initializing Static and Global Variables in TI ARM9 ABI and TIABI Modes 107
5.15 Changing the ANSI/ISO C Language Mode 108
5.16 GNU Language Extensions 110
5.17 AUTOSAR 113
5.18 Compiler Limits 113

Copyright © 2012, Texas Instruments Incorporated

5.1 Characteristics of ARM C

The compiler supports the C language as defined by ISO/IEC 9899:1990, which is equivalent to American National Standard for Information Systems-Programming Language C X3.159-1989 standard, commonly referred to as C89, published by the American National Standards Institute. The compiler can also accept many of the language extensions found in the GNU C compiler (see Section 5.16). The compiler does not support C99.

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

Unsupported features of the C library are:

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

5.2 Characteristics of ARM C++

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

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

The *exceptions* to the standard are as follows:

- The compiler does not support embedded C++ run-time-support libraries.
- The library supports wide chars (wchar_t), in that template functions and classes that are defined for char are also available for wchar_t. For example, wide char stream classes wios, wiostream, wstreambuf and so on (corresponding to char classes ios, iostream, streambuf) are implemented. However, there is no low-level file I/O for wide chars. Also, the C library interface to wide char support (through the C++ headers <cwchar> and <cwctype>) is limited as described above in the C library.
- For TIABI (deprecated) only: If the definition of an inline function contains a static variable, and it appears in multiple compilation units (usually because it's a member function of a class defined in a header file), the compiler generates multiple copies of the static variable rather than resolving them to a single definition. The compiler emits a warning (#1369) in such cases.
- Two-phase name binding in templates, as described in [tesp.res] and [temp.dep] of the standard, is not implemented.
- The export keyword for templates is not implemented.
- A typedef of a function type cannot include member function cv-qualifiers.
- A partial specialization of a class member template cannot be added outside of the class definition.

Copyright © 2012, Texas Instruments Incorporated

5.3 Using MISRA-C:2004

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

- The --check_misra option enables checking of the specified MISRA-C:2004 rules.
- The CHECK_MISRA pragma enables/disables MISRA-C:2004 rules at the source level. This pragma is equivalent to using the --check_misra option. See Section 5.9.1.
- RESET_MISRA pragma resets the specified MISRA-C:2004 rules to the state they were before any CHECK_MISRA pragmas were processed. See Section 5.9.15.

The syntax of the option and pragmas is:

```
--check_misra={all|required|advisory|none|rulespec}
#pragma CHECK_MISRA ("{all|required|advisory|none|rulespec}");
#pragma RESET_MISRA ("{all|required|advisory|rulespec}");
```

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

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

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

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

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

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

The syntax for these options is:

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


Copyright © 2012, Texas Instruments Incorporated

5.4 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.

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

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

(1) In TIARM9 ABI mode, 64-bit data is aligned on a 32-bit boundary. In EABI mode, 64-bit data is aligned on a 64-bit boundary.
(2) Figures are minimum precision.

In EABI mode, 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. EABI 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  }
```


Copyright © 2012, Texas Instruments Incorporated

For example, the following code results in an enumerator type of short:

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

5.5 Keywords

The ARM C/C++ compiler supports the standard const and volatile keywords. In addition, the C/C++ compiler extends the C/C++ language through the support of the interrupt keyword.

5.5.1 The const Keyword

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

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

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

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

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

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

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

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

5.5.2 The interrupt Keyword

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

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

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

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

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

Copyright © 2012, Texas Instruments Incorporated

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

HWI Objects and the interrupt Keyword

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

5.5.3 The volatile Keyword

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

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

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

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

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

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

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

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

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

Example 5-1. Volatile for Local Variables With setjmp

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


Copyright © 2012, Texas Instruments Incorporated

5.6 C++ Exception Handling

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

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

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

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

Using --exceptions causes the compiler to insert exception handling code. This code will increase the code size of the program, particularly for TI_ARM9_ABI and TIABI (deprecated) modes. In addition, TI_ARM9_ABI and TIABI will increase the execution time, even if an exception is never thrown. EABI will not increase code size as much, and has a minimal execution time cost if exceptions are never thrown, but will slightly increase the data size for the exception-handling tables.

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

Copyright © 2012, Texas Instruments Incorporated

5.7 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.

5.7.1 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.

5.7.2 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.

Copyright © 2012, Texas Instruments Incorporated

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.

5.8 The asm Statement

The C/C++ compiler can embed assembly language instructions or directives directly into the assembly language output of the compiler. This capability is an extension to the C/C++ language—the *asm* statement. The asm (or __asm) statement provides access to hardware features that C/C++ cannot provide. The asm statement is syntactically like a call to a function named asm, with one string constant argument:

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

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

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

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

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

Copyright © 2012, Texas Instruments Incorporated

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

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

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

5.9 Pragma Directives

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

- CHECK_MISRA (See Section 5.9.1)
- CLINK (See Section 5.9.2)
- CODE_SECTION (See Section 5.9.3)
- CODE_STATE (See Section 5.9.4)
- DATA_ALIGN (See Section 5.9.5)
- DATA_SECTION (See Section 5.9.6)
- DIAG_SUPPRESS, DIAG_REMARK, DIAG_WARNING, DIAG_ERROR, and DIAG_DEFAULT (See Section 5.9.7)
- DUAL_STATE (See Section 5.9.8)
- FUNC_EXT_CALLED (See Section 5.9.9)
- FUNCTION_OPTIONS (See Section 5.9.10)
- INTERRUPT (See Section 5.9.11)
- LOCATION (EABI only; see Section 5.9.12)
- MUST_ITERATE (See Section 5.9.13)
- NO_HOOKS (See Section 5.9.14)
- RESET_MISRA (See Section 5.9.15)
- RETAIN (See Section 5.9.16)
- SET_CODE_SECTION (See Section 5.9.17)
- SET_DATA_SECTION (See Section 5.9.17)
- SWI_ALIAS (See Section 5.9.18)
- TASK (See Section 5.9.19)
- UNROLL (See Section 5.9.20)
- WEAK (See Section 5.9.21)

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

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

Copyright © 2012, Texas Instruments Incorporated

5.9.1 The CHECK_MISRA Pragma

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

The syntax of the pragma in C is:

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

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

The RESET_MISRA pragma can be used to reset any CHECK_MISRA pragmas; see Section 5.9.15.

5.9.2 The CLINK Pragma

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

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

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

The RETAIN pragma has the opposite effect of the CLINK pragma. See Section 5.9.16 for more details.

5.9.3 The CODE_SECTION Pragma

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

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

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

The following examples demonstrate the use of the CODE_SECTION pragma.

Example 5-2. Using the CODE_SECTION Pragma C Source File

```
#pragma CODE_SECTION(fn, "my_sect")

int fn(int x)
{
    return x;
}
```


Copyright © 2012, Texas Instruments Incorporated

Example 5-3. Generated Assembly Code From Example 5-2

```
            .sect     "my_sect"
            .align    4
            .clink
            .armfunc fn
            .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
```

5.9.4 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* **);**

5.9.5 The DATA_ALIGN Pragma

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

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

Copyright © 2012, Texas Instruments Incorporated

5.9.6 The DATA_SECTION Pragma

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

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

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

Example 5-4 through Example 5-6 demonstrate the use of the DATA_SECTION pragma.

Example 5-4. Using the DATA_SECTION Pragma C Source File

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

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

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

Example 5-6. Using the DATA_SECTION Pragma Assembly Source File

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


Copyright © 2012, Texas Instruments Incorporated

5.9.7 The Diagnostic Message Pragmas

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

Pragma	Option	Description
DIAG_SUPPRESS *num*	-pds=*num*[, num_2, num_3...]	Suppress diagnostic *num*
DIAG_REMARK *num*	-pdsr=*num*[, num_2, num_3...]	Treat diagnostic *num* as a remark
DIAG_WARNING *num*	-pdsw=*num*[, num_2, num_3...]	Treat diagnostic *num* as a warning
DIAG_ERROR *num*	-pdse=*num*[, num_2, num_3...]	Treat diagnostic *num* as an error
DIAG_DEFAULT *num*	n/a	Use default severity of the diagnostic

The syntax of the pragmas in C is:

#pragma DIAG_*XXX* [=]*num*[, num_2, num_3...]

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

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

5.9.8 The 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 6.10.

Copyright © 2012, Texas Instruments Incorporated

5.9.9 The FUNC_EXT_CALLED Pragma

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

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

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

The syntax of the pragma in C is:

#pragma FUNC_EXT_CALLED (*func* **);**

The syntax of the pragma in C++ is:

#pragma FUNC_EXT_CALLED;

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

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

5.9.10 The FUNCTION_OPTIONS Pragma

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

The syntax of the pragma in C is:

#pragma FUNCTION_OPTIONS (*func*, **"***additional options***");**

The syntax of the pragma in C++ is:

#pragma FUNCTION_OPTIONS("*additional options***");**

5.9.11 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* **)**] **;**

Copyright © 2012, Texas Instruments Incorporated

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).

HWI Objects and the INTERRUPT Pragma

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

5.9.12 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 attribute. Location support is only available in EABI.

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 attribute is:

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

5.9.13 The MUST_ITERATE Pragma

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

Copyright © 2012, Texas Instruments Incorporated

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

5.9.13.1 The MUST_ITERATE Pragma Syntax

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

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

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

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

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

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

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

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

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

5.9.13.2 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. For example:

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

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

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

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

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


Copyright © 2012, Texas Instruments Incorporated

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

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

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

5.9.14 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.

5.9.15 The RESET_MISRA Pragma

The RESET_MISRA pragma resets the specified MISRA-C:2004 rules to the state they were before any CHECK_MISRA pragmas (see Section 5.9.1) were processed. For instance, if a rule was enabled on the command line but disabled in the source, the RESET_MISRA pragma resets it to enabled. This pragma accepts the same format as the --check_misra option, except for the "none" keyword.

The syntax of the pragma in C is:

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

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

5.9.16 The RETAIN Pragma

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

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

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

The CLINK pragma has the opposite effect of the RETAIN pragma. See Section 5.9.2 for more details.

Copyright © 2012, Texas Instruments Incorporated

5.9.17 The SET_CODE_SECTION and SET_DATA_SECTION Pragmas

These pragmas can be used to set the section for all declarations below the pragma.

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

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

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

In Example 5-7 x and y are put in the section mydata. To reset the current section to the default used by the compiler, a blank paramater should be passed to the pragma. An easy way to think of the pragma is that it is like applying the CODE_SECTION or DATA_SECTION pragma to all symbols below it.

Example 5-7. Setting Section With SET_DATA_SECTION Pragma

```
#pragma SET_DATA_SECTION("mydata")
int x;
int y;
#pragma SET_DATA_SECTION()
```

The pragmas apply to both declarations and definitions. If applied to a declaration and not the definition, the pragma that is active at the declaration is used to set the section for that symbol. Here is an example:

Example 5-8. Setting a Section With SET_CODE_SECTION Pragma

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

In Example 5-8 func1 is placed in section func1. If conflicting sections are specified at the declaration and definition, a diagnostic is issued.

The current CODE_SECTION and DATA_SECTION pragmas and GCC attributes can be used to override the SET_CODE_SECTION and SET_DATA_SECTION pragmas. For example:

Example 5-9. Overriding SET_DATA_SECTION Setting

```
#pragma DATA_SECTION(x, "x_data")
#pragma SET_DATA_SECTION("mydata")
int x;
int y;
#pragma SET_DATA_SECTION()
```

In Example 5-9 x is placed in x_data and y is placed in mydata. No diagnostic is issued for this case.

The pragmas work for both C and C++. In C++, the pragmas are ignored for templates and for implictly created objects, such as implicit constructors and virtual function tables.

Copyright © 2012, Texas Instruments Incorporated

5.9.18 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 more information about using software interrupts, including restrictions on passing arguments and register usage, see Section 6.6.5.

Example 5-10. 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.                          */
}
```

Example 5-11. 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
```


Copyright © 2012, Texas Instruments Incorporated

5.9.19 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

5.9.20 The UNROLL Pragma

The UNROLL pragma specifies to the compiler how many times a loop should be unrolled. The optimizer must be invoked (use --opt_level=[1|2|3] or -O1, -O2, or -O3) in order for pragma-specified loop unrolling to take place. The compiler has the option of ignoring this pragma.

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

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

#pragma UNROLL(*n* **);**

If possible, the compiler unrolls the loop so there are *n* copies of the original loop. The compiler only unrolls if it can determine that unrolling by a factor of *n* is safe. In order to increase the chances the loop is unrolled, the compiler needs to know certain properties:

- The loop iterates a multiple of *n* times. This information can be specified to the compiler via the multiple argument in the MUST_ITERATE pragma.
- The smallest possible number of iterations of the loop
- The largest possible number of iterations of the loop

The compiler can sometimes obtain this information itself by analyzing the code. However, sometimes the compiler can be overly conservative in its assumptions and therefore generates more code than is necessary when unrolling. This can also lead to not unrolling at all.

Furthermore, if the mechanism that determines when the loop should exit is complex, the compiler may not be able to determine these properties of the loop. In these cases, you must tell the compiler the properties of the loop by using the MUST_ITERATE pragma.

Specifying #pragma UNROLL(1); asks that the loop not be unrolled. Automatic loop unrolling also is not performed in this case.

If multiple UNROLL pragmas are specified for the same loop, it is undefined which pragma is used, if any.

Copyright © 2012, Texas Instruments Incorporated

5.9.21 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.

5.10 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.

Copyright © 2012, Texas Instruments Incorporated

5.11 Application Binary Interface

Selecting one of the three ABIs supported by the ARM compiler is discussed in Section 2.13.

An ABI should define how functions that are written separately, and compiled or assembled separately can work together. This involves standardizing the data type representation, register conventions, and function structure and calling conventions. It should define linkname generation from C symbol names. 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.

5.11.1 TI_ARM9_ABI

The TI ARM9 ABI is the default mode of the compiler. This ABI is documented throughout this document and the *Assembly Language Tools User's Guide*. Refer to the following sections for more information:

Section 5.4	Data Types
Section 6.2	Object Representation
Section 6.3	Register Conventions
Section 6.4	Function Structure and Calling Conventions
Section 6.9	System Initialization

In the TI ARM9 ABI, the link step generates any necessary code when cross-mode calls are made. No extra code needs to be written to handle the transfer from arm-mode to thumb-mode, or vice-versa. However, all 32-bit and 16-bit assembly routines must be explicitly identified using the .armfunc and .thumbfunc assembler directives. These directives are required for the link step to properly generate dual mode code. Assembly code without these directives may result in compiler errors and incorrect execution. Refer to the *Assembly Language Tools User's Guide* for details on the .armfunc and .thumbfunc directives.

The __TI_ARM9ABI_ASSEMBLER predefined symbol is set to 1 if compiling for TI ARM9 ABI and is set to 0 otherwise. To write code that can be used with multiple ABIs, the assembly code must use this predefined symbol to make the code conditional.

All function names are prefixed with the '_' (underscore) for both 16-bit mode (thumb) or 32-bit mode (arm). If you are writing hand-coded assembly routines, you must use only the underscore prefix.

For example, if function (foo) is to be defined in 16-bit mode, and the code needs to be assembled for TI ARM9 ABI and TI ABI, this is the desired assembly sequence:

```
        .if __TI_ARM9ABI_ASSEMBLER
        .thumbfunc _foo
        .endif

.if __TIARM9ABI_ASSEMBLER
_foo:
.else
$foo:
.endif
```

5.11.2 TIABI (Deprecated)

The TIABI mode has been deprecated; it is recommended that those still using this ABI move to TI_ARM9_ABI.

The TIABI mode is the ABI of older TI 2.x ARM compilers. You must enable this ABI (--abi=tiabi) if you are linking object code or object libraries that have been compiled with an older compiler. You must also enable this ABI if you are compiling hand-coded assembly source files that were initially written for a TI 2.x ARM compiler. The TIABI mode cannot be used with any architecture option for ARM9E or higher.

The only difference between TIABI mode and TI ARM9 ABI mode is that in TIABI the link step cannot generate the code to support cross-mode calls. The compiler generates veneers to support-cross mode calls. Refer to Section 6.10 for more information.

Copyright © 2012, Texas Instruments Incorporated

5.11.3 ARM ABIv2 or EABI

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 and the documentation is a reference point for those writing utilities that require an understanding of the ABI. The complete ARM ABI specifications can be found at http://www.arm.com/products/DevTools/ABI.html.
- 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 6.9.3.

The __TI_EABI_ASSEMBLER predefined symbol is set to 1 if compiling for EABI and is set to 0 otherwise.

5.12 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.

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
_ _ldrex		yes			yes	yes	yes
_ _ldrexb		yes			yes	yes	yes
_ _ldrexd						yes	yes
_ _ldrexh		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
_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
_shadd16		yes			yes	yes	yes
_shadd8		yes			yes	yes	yes

Copyright © 2012, Texas Instruments Incorporated

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)
_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
_smusadx		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
_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

Copyright © 2012, Texas Instruments Incorporated

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)
_ _strexb		yes			yes	yes	yes
_ _strexd						yes	yes
_ _strexh		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
_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

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.

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
unsigned int *dest* = _ _**ldrex(**void* *src* **);**	**LDREX** *dst* **,** *src*	Loads data from memory address containing word (32-bit) data

Copyright © 2012, Texas Instruments Incorporated

Table 5-4. ARM Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
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
unsigned int *dest* = _ _**ldrexh(**void* *src* **);**	**LDREXH** *dst* **,** *src*	Loads data from memory address containing halfword (16-bit) data
int *dst* = **_norm(**int *src* **);**	**CLZ** *dst* **,** *src*	Normalize floating point
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
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.
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

Copyright © 2012, Texas Instruments Incorporated

Table 5-4. ARM Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
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 *src1* , int *src2* **);**	**SMULBB** *tmp* , *src1* , *src2* **QDADD** *dst* , *dst* , *tmp*	Saturated multiply-accumulate
int *dst* =**_smlabb(**int *src1* , int *src2* , int *acc* **);**	**SMLABB** *dst* , *src1* , *src2*	Signed multiply-accumulate bottom halfwords
int *dst* =**_smlabt(**int *src1* , int *src2* , int *acc* **);**	**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(**int *src1* , int *src2* **);**	**SMLALBB** *dstlo* , *dsthi* , *src1* , *src2*	Signed multiply-long and accumulate bottom halfwords
long long *dst* =**_smlalbt(**int *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(**int *src1* , int *src2* **);**	**SMLALTB** *dstlo* , *dsthi* , *src1* , *src2*	Signed multiply-long and accumulate top and bottom halfwords
long long *dst* =**_smlaltt(**int *src1* , int *src2* **);**	**SMLALTT** *dstlo* , *dsthi* , *src1* , *src2*	Signed multiply-long and accumulate top halfwords
int *dst* =**_smlatb(**int *src1* , int *src2* , int *acc* **);**	**SMLATB** *dst* , *src1* , *src2*	Signed multiply-accumulate top and bottom halfwords
int *dst* =**_smlatt(**int *src1* , int *src2* , int *acc* **);**	**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.
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.

Copyright © 2012, Texas Instruments Incorporated

Table 5-4. ARM Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
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* **,** int *acc* **);**	**SMUAD** *dst* **,** *src1* **,** *src2* **,** *acc*	Performs two signed 16-bit multiplications on the top and bottom halfwords and adds the products.
int *dst* **_smuadx(**int *src1* **,** int *src2* **,** int *acc* **);**	**SMUADX** *dst* **,** *src1* **,** *src2* **,** *acc*	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* **_smusadx(**int *src1* **,** int *src2* **,** int *acc* **);**	**SMUSDX** *dst* **,** *src1* **,** *src2* **,** *acc*	Same as _smusd except the halfwords in src2 are exchange before the multiplication.
int *dst* **_smusd(**int *src1* **,** int *src2* **,** int *acc* **);**	**SMUSD** *dst* **,** *src1* **,** *src2* **,** *acc*	Performs two signed 16-bit multiplications on the top and bottom halfwords and subtracts the products.
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.

Copyright © 2012, Texas Instruments Incorporated

Table 5-4. ARM Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
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
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

Copyright © 2012, Texas Instruments Incorporated

Table 5-4. ARM Compiler Intrinsics (continued)

C/C++ Compiler Intrinsic	Assembly Instruction	Description
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.

5.13 Object File Symbol Naming Conventions (Linknames)

Each externally visible identifier is assigned a unique symbol name to be used in the object file, a so-called *linkname*. This name is assigned by the compiler according to an algorithm which depends on the name, type, and source language of the symbol. This algorithm may add a prefix to the identifier (typically an underscore), and it may *mangle* the name. This algorithm may *mangle* the name.

In TIABI (deprecated), the linkname for all objects is the same as the name in the C source with an added underscore prefix. For C functions, an underscore is prefixed for 32-bit mode and 16-bit mode functions with the TI ARM9 ABI (--abi=ti_arm9_abi) mode. In the backward-compatible TIABI mode (--abi=tiabi), the names of 32-bit mode functions are prefixed with an underscore and 16-bit mode functions are prefixed with a dollar sign ($). This prevents any C identifier from colliding with any identifier in the assembly code namespace, such as an assembler keyword.

In EABI, no prefix is used. If a C identifier would collide with an assembler keyword, the compiler will escape 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.

C++ functions have the same initial character: an underscore for the TI ARM9 ABI, or an underscore or dollar sign for the TIABI. Additionally, the function name is mangled further. 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 TI ARM9 ABI or TIABI, the mangling algorithm used closely follows that described in The Annotated Reference Manual (ARM).

For example, the general form of a C++ linkname for a 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
```


Copyright © 2012, Texas Instruments Incorporated

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.

For EABI, 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"

> **EABI Mode C++ Demangling**
>
> **NOTE:** The EABI mode has a different C++ demangling scheme. For instance, there is no prefix (either _ or $). Please refer to http://www.arm.com/products/DevTools/ABI.html for details.

5.14 Initializing Static and Global Variables in TI ARM9 ABI and TIABI Modes

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

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

5.14.1 Initializing Static and Global Variables With the Linker

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

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

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

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

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

5.14.2 Initializing Static and Global Variables With the const Type Qualifier

Static and global variables of type *const* without explicit initializations are similar to other static and global variables because they might not be preinitialized to 0 (for the same reasons discussed in Section 5.14). For example:

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

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

Copyright © 2012, Texas Instruments Incorporated

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

This corresponds to an entry in the .const section:

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

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

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

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

is compiled into this assembly code:

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

5.15 Changing the ANSI/ISO C Language Mode

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

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

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

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

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

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

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

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

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

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

```
unsigned short u;
int i;
```


Copyright © 2012, Texas Instruments Incorporated

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Use the --relaxed_ansi option when you want the compiler to ignore strict ANSI/ISO violations rather than emit a warning (as occurs in normal ANSI/ISO mode) or an error message (as occurs in strict ANSI/ISO mode). In relaxed ANSI/ISO mode, the compiler accepts extensions to the ANSI/ISO C standard, even when they conflict with ANSI/ISO C. The GCC language extensions described in Section 5.16 are available in relaxed ANSI/ISO mode.

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

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

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

Embedded C++ omits these C++ features:

Copyright © 2012, Texas Instruments Incorporated

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

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

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

5.16 GNU Language Extensions

The GNU compiler collection (GCC) defines a number of language features not found in the ANSI/ISO C and C++ standards. The definition and examples of these extensions (for GCC version 3.4) can be found at the GNU web site, http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/C-Extensions.html.

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

5.16.1 Extensions

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

The extensions that the TI compiler supports are listed in Table 5-5, 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
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

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

Copyright © 2012, Texas Instruments Incorporated

Table 5-5. GCC Language Extensions (continued)

Extensions	Descriptions
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
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 5.16.5)
Vector extensions	Using vector instructions through built-in functions
Target built-ins	Built-in functions specific to particular targets
Pragmas	Pragmas accepted by GCC
Unnamed fields	Unnamed struct/union fields within structs/unions
Thread-local	Per-thread variables

5.16.2 Function Attributes

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

In addition, the visibility function attribute is supported for EABI mode (--abi=eabi).

The format attribute is applied to the declarations of printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf, scanf, fscanf, vfscanf, vscanf, vsscanf, and sscanf in stdio.h. Thus when GCC extensions are enabled, the data arguments of these functions are type checked against the format specifiers in the format string argument and warnings are issued when there is a mismatch. These warnings can be suppressed in the usual ways if they are not desired.

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

5.16.3 Variable Attributes

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

The used attribute is defined in GCC 4.2 (see http://gcc.gnu.org/onlinedocs/gcc-4.2.4/gcc/Variable-Attributes.html#Variable-Attributes).

In addition, the weak variable attribute is supported for EABI mode (--abi=eabi).

Copyright © 2012, Texas Instruments Incorporated

5.16.4 Type Attributes

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

In addition, the visibility type attribute is supported for EABI mode (--abi=eabi).

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

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

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

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

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

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

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

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

Limitation on packed for COFF ABI

NOTE: Packed structs with 64-bit scalar members are not supported on ARM non-Cortex versions with COFF ABI.

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*.

5.16.5 Built-In Functions

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

The __builtin_frame_address function returns zero unless the argument is a constant zero.

The __builtin_return_address function always returns zero.

Copyright © 2012, Texas Instruments Incorporated

5.17 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.

5.18 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.

Copyright © 2012, Texas Instruments Incorporated

TEXAS INSTRUMENTS

Chapter 6
SPNU151H–August 2012

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.

Topic **Page**

6.1 Memory Model 115
6.2 Object Representation 117
6.3 Register Conventions 124
6.4 Function Structure and Calling Conventions 127
6.5 Interfacing C and C++ With Assembly Language 130
6.6 Interrupt Handling 134
6.7 Intrinsic Run-Time-Support Arithmetic and Conversion Routines 137
6.8 Built-In Functions 140
6.9 System Initialization 140
6.10 Dual-State Interworking Under TIABI (Deprecated) 151

Copyright © 2012, Texas Instruments Incorporated

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.

The Linker Defines the Memory Map

NOTE: The linker, not the compiler, defines the memory map and allocates code and data into target memory. The compiler assumes nothing about the types of memory available, about any locations not available for code or data (holes), or about any locations reserved for I/O or control purposes. The compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces.

For example, you can use the linker to allocate global variables into on-chip RAM or to allocate executable code into external ROM. You can allocate each block of code or data individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although you can access physical memory locations with C/C++ pointer types).

6.1.1 Sections

The compiler produces relocatable blocks of code and data called *sections*. The sections are allocated into memory in a variety of ways to conform to a variety of system configurations. For more information about sections and allocating them, see the introductory object file information in the *ARM Assembly Language Tools User's Guide*.

There are two basic types of sections:

- **Initialized sections** contain data or executable code. The C/C++ compiler creates the following initialized sections:
  - For TI_ARM9_ABI and TIABI (deprecated), the **.cinit section** contains tables for initializing variables and constants.
  - The **.pinit section** for TI_ARM9_ABI and TIABI, or the **.init_array section** for EABI, contains global constructor tables.
  - For EABI only, the **.data section** contains initialized global and static variables.
  - The **.const section** contains string constants and data defined with the C/C++ qualifier *const* (provided the constant is not also defined as *volatile*).
  - The **.text section** contains all the executable code. It also contains string literals, switch tables, and compiler-generated constants. 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.
- **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 TI_ARM9_ABI and TIABI, the **.bss section** reserves space for global and static variables. At boot or load time, the C/C++ boot routine or the loader copies data out of the .cinit section (which can be in ROM) and stores it in the .bss section.
  - For EABI only, the **.bss section** reserves space for uninitialized global and static variables.
  - The **.stack section** reserves memory for the C/C++ software stack.
  - The **.sysmem section** reserves space for dynamic memory allocation. The reserved space is used by dynamic memory allocation routines, such as malloc, calloc, realloc, or new. If a C/C++ program does not use these functions, the compiler does not create the .sysmem section.

The assembler creates the default sections .text, .bss, and .data. The C/C++ compiler, however, does not use the .data section. You can instruct the compiler to create additional sections by using the CODE_SECTION and DATA_SECTION pragmas (see Section 5.9.3 and Section 5.9.6).

Copyright © 2012, Texas Instruments Incorporated

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	ROM or 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, __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 if 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.

Stack Overflow

NOTE: The compiler provides no means to check for stack overflow during compilation or at run time. A stack overflow disrupts the run-time environment, causing your program to fail. Be sure to allow enough space for the stack to grow. You can use the --entry_hook option to add code to the beginning of each function to check for stack overflow; see Section 2.15.

Copyright © 2012, Texas Instruments Incorporated

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, __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*.

Dynamically allocated objects are not addressed directly (they are always accessed with pointers) and the memory pool is in a separate section (.sysmem); therefore, the dynamic memory pool can have a size limited only by the amount of available memory in your system. To conserve space in the .bss section, you can allocate large arrays from the heap instead of defining them as global or static. For example, instead of a definition such as:

```
struct big table[100];
```

Use a pointer and call the malloc function:

```
struct big *table
table = (struct big *)malloc(100*sizeof(struct big));
```

6.1.4 Initialization of Variables in TI_ARM9_ABI and TIABI

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

In situations where a program is loaded directly from an object file into memory and run, you can avoid having the .cinit section occupy space in memory. A loader can read the initialization tables directly from the object file (instead of from ROM) and perform the initialization directly at load time instead of at run time. You can specify this to the linker by using the --ram_model link option. For more information, see Section 6.9.

6.2 Object Representation

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
enum (TI_ARM9_ABI and TIABI only. [(2)].)	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[(3)]
unsigned long long	Even/odd register pair	64 bits aligned to 32-bit (word) boundary[(3)]

(1) Negative values are sign-extended to bit 31.
(2) For enum information for EABI mode, see Table 5-2.
(3) In TIARM9 ABI mode, 64-bit data is aligned on a 32-bit boundary. In EABI mode, 64-bit data is aligned on a 64-bit boundary.

Copyright © 2012, Texas Instruments Incorporated

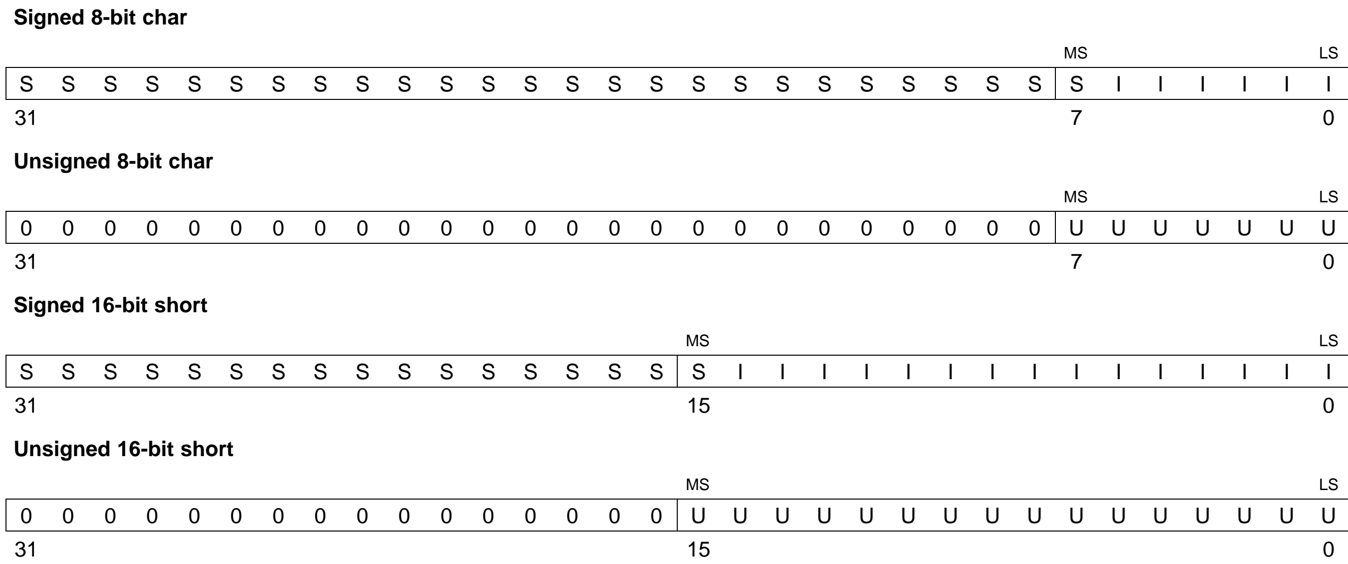
Table 6-2. Data Representation in Registers and Memory (continued)

Data Type	Register Storage	Memory Storage
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[3]
long double	Register pair	64 bits aligned to 32-bit (word) boundary[3]
struct	Members are stored as their individual types require.	Members are stored as their individual types require; aligned according to the member with the most restrictive alignment requirement.
array	Members are stored as their individual types require.	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 their individual types require	64 bits aligned to 32-bit (word) boundary

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



LEGEND: S = sign, I = signed integer, U = unsigned integer, MS = most significant, LS = least significant

Copyright © 2012, Texas Instruments Incorporated

6.2.1.2 enum (TI_ARM9_ABI or TIABI), float, int, and long Data Types (signed and unsigned)

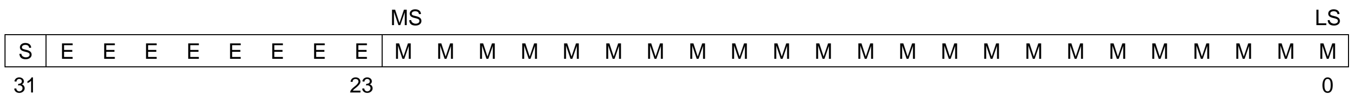
The int, unsigned int, enum, float, long and unsigned long data types are stored in memory as 32-bit objects at word (4 byte) aligned addresses. Objects of these types are loaded to and stored from bits 0-31 of a register, as shown in Figure 6-2. In big-endian mode, 4-byte objects are loaded to registers 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.

Enums in EABI Mode

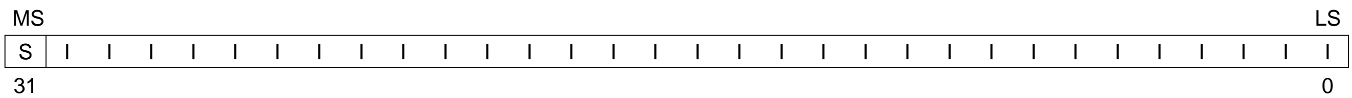
NOTE: Enumerations have a different representation in EABI mode; see Table 5-2.

Figure 6-2. 32-Bit Data Storage Format

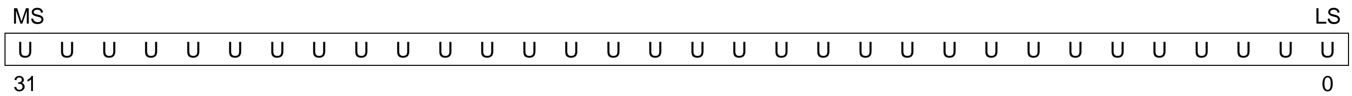
Single-precision floating char



Signed 32-bit integer, enum, or long char



Unsigned 32-bit integer or long



LEGEND: S = sign, M = Mantissa, U = unsigned integer, E = exponent, I = signed integer, MS = most significant, LS = least significant

Copyright © 2012, Texas Instruments Incorporated

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 Figure 6-3. 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

Address *x*

												MS																		
S	E	E	E	E	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
31											20																				0

Address *x*+ 4

																															LS
M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
31																															0

LEGEND: S = sign, M = mantissa, E = exponent, MS = most significant, LS = least significant

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 nonvirtual. The parameter f is the pointer to the member function if it is nonvirtual, when I is 0. The 0 is the offset to the virtual function pointer within the class object.

6.2.1.6 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.

Copyright © 2012, Texas Instruments Incorporated

6.2.2 Bit Fields

Bit fields are handled differently in TI ARM9 ABI and TIABI, and EABI modes. Section 6.2.2.1 details how bit fields are handled in all modes. Section 6.2.2.2 details how bit fields differ in EABI mode.

6.2.2.1 Generic Bit Fields

Bit fields are the only objects that are packed within a byte. That is, two bit fields can be stored in the same byte. Bit fields can range in size from 1 to 32 bits, but they never span a 4-byte boundary.

For big-endian mode, bit fields are packed into registers from most significant bit (MSB) to least significant bit (LSB) in the order in which they are defined. Bit fields are packed in memory from most significant byte (MSbyte) to least significant byte (LSbyte). For little-endian mode, bit fields are packed into registers from the LSB to the MSB in the order in which they are defined, and packed in memory from LSbyte to MSbyte.

Figure 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

MS LS

A	A	A	A	A	A	A	B	B	B	B	B	B	B	B	B	B	C	C	C	D	D	E	E	E	E	E	E	E	E	E	X
6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	2	1	0	1	0	8	7	6	5	4	3	2	1	0	X

31 0

Big-endian memory

Byte 0								Byte 1								Byte 2								Byte 3							
A	A	A	A	A	A	A	B	B	B	B	B	B	B	B	B	B	C	C	C	D	D	E	E	E	E	E	E	E	E	E	X
6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	2	1	0	1	0	8	7	6	5	4	3	2	1	0	X

Little-endian register

MS LS

X	E	E	E	E	E	E	E	E	E	D	D	C	C	C	B	B	B	B	B	B	B	B	B	B	A	A	A	A	A	A	A
X	8	7	6	5	4	3	2	1	0	1	0	2	1	0	9	8	7	6	5	4	3	2	1	0	6	5	4	3	2	1	0

31 0

Little-endian memory

Byte 0								Byte 1								Byte 2								Byte 3							
B	A	A	A	A	A	A	A	B	B	B	B	B	B	B	B	E	E	D	D	C	C	C	B	X	E	E	E	E	E	E	E
0	6	5	4	3	2	1	0	8	7	6	5	4	3	2	1	1	0	1	0	2	1	0	9	X	8	7	6	5	4	3	2

LEGEND: X = not used, MS = most significant, LS = least significant

Copyright © 2012, Texas Instruments Incorporated

6.2.2.2 EABI Bit Field Differences

Bit fields are handled differently in TI ARM9 ABI and TIABI modes versus EABI mode in these ways:

- In TI ARM9 ABI and TIABI modes, plain int bit fields are signed. In EABI they are unsigned. Consider the following C code:

```
struct st
{
   int a:5;
} S;

foo()
{
   if (S.a < 0)
      bar();
}
```

  In EABI bar () is never called as bit field 'a' is unsigned. Use signed int if you need a signed bit field in EABI mode.
- In TI ARM9 ABI and TIABI modes, bit fields of type long long are not allowed. In EABI, long long bit fields are supported.
- In TI ARM9 ABI and TIABI modes, all bit fields are treated as signed or unsigned int type. In EABI, bit fields are treated as the declared type.
- In TI ARM9 ABI and TIABI modes, the size and alignment a bit field contributes to the struct containing it depends on the number of bits in the bit field. In EABI, 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};
```

  In TI ARM9 ABI and TIABI modes, this struct takes up 1 byte and is aligned at 1 byte. In EABI, this struct uses up 4 bytes and is aligned at 4 bytes.
- In TI ARM9 ABI and TIABI modes, unnamed bit fields are zero-sized bit fields do not affect the struct or union alignment. In EABI, such fields affect the alignment of the struct or union. For example, consider the struct:

```
struct st{char a:4; int :22;};
```

  In TI ARM9 ABI and TIABI modes, this struct uses 4 bytes and is aligned at a 1-byte boundary. In EABI, this struct uses 4 bytes and is aligned at a 4-byte boundary.
- With EABI, 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.

Copyright © 2012, Texas Instruments Incorporated

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 6.9.

- In an expression. For example:

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

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

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

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

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

  Because strings are stored in the .const section (possibly in ROM) and shared, it is bad practice for a program to modify a string constant. The following code is an example of incorrect string use:

```
const char   *a = "abc"
a[1] = 'x';             /* Incorrect! */
```


Copyright © 2012, Texas Instruments Incorporated

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. Table 6-3 shows the types of registers affected by these conventions.Table 6-4 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

(1) The parent function refers to the function making the function call. The child function refers to the function being called.
(2) The SP is preserved by the convention that everything pushed on the stack is popped off before returning.

Copyright © 2012, Texas Instruments Incorporated

Table 6-5. VFP Register Usage

32-Bit Register	64-Bit Register	Usage	Preserved by Function [(1)]
FPSCR		Status register	N/A
S0 S1	D0	Floating-point expression, return values, pass arguments	N/A
S2 S3	D1	Floating-point expression, return values, pass arguments	N/A
S4 S5	D2	Floating-point expression, return values, pass arguments	N/A
S6 S7	D3	Floating-point expression, return values, pass arguments	N/A
S8 S9	D4	Floating-point expression, pass arguments	N/A
S10 S11	D5	Floating-point expression, pass arguments	N/A
S12 S13	D6	Floating-point expression, pass arguments	N/A
S14 S15	D7	Floating-point expression, pass arguments	N/A
S16 S17	D8	Floating-point expression	Child
S18 S19	D9	Floating-point expression	Child
S20 S21	D10	Floating-point expression	Child
S22 S23	D11	Floating-point expression	Child
S24 S25	D12	Floating-point expression	Child
S26 S27	D13	Floating-point expression	Child
S28 S29	D14	Floating-point expression	Child
S30 S31	D15	Floating-point expression	Child
	D16-D31	Floating-point expression	

[(1)] The child function refers to the function being called.

Copyright © 2012, Texas Instruments Incorporated

Table 6-6. Neon Register Usage

64-Bit Register	Quad Register	Usage	Preserved by Function [(1)]
D0 D1	Q0	SIMD register	N/A
D2 D3	Q1	SIMD register	N/A
D4 D5	Q2	SIMD register	N/A
D6 D7	Q3	SIMD register	N/A
D8 D9	Q4	SIMD register	Child
D10 D11	Q5	SIMD register	Child
D12 D13	Q6	SIMD register	Child
D14 D15	Q7	SIMD register	Child
D16 D17	Q8	SIMD register	N/A
D18 D19	Q9	SIMD register	N/A
D20 D21	Q10	SIMD register	N/A
D22 D23	Q11	SIMD register	N/A
D24 D25	Q12	SIMD register	N/A
D26 D27	Q13	SIMD register	N/A
D28 D29	Q14	SIMD register	N/A
D30 D31	Q15	SIMD register	N/A
FPSCR		Status register	N/A

(1) The child function refers to the function being called.

Copyright © 2012, Texas Instruments Incorporated

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:

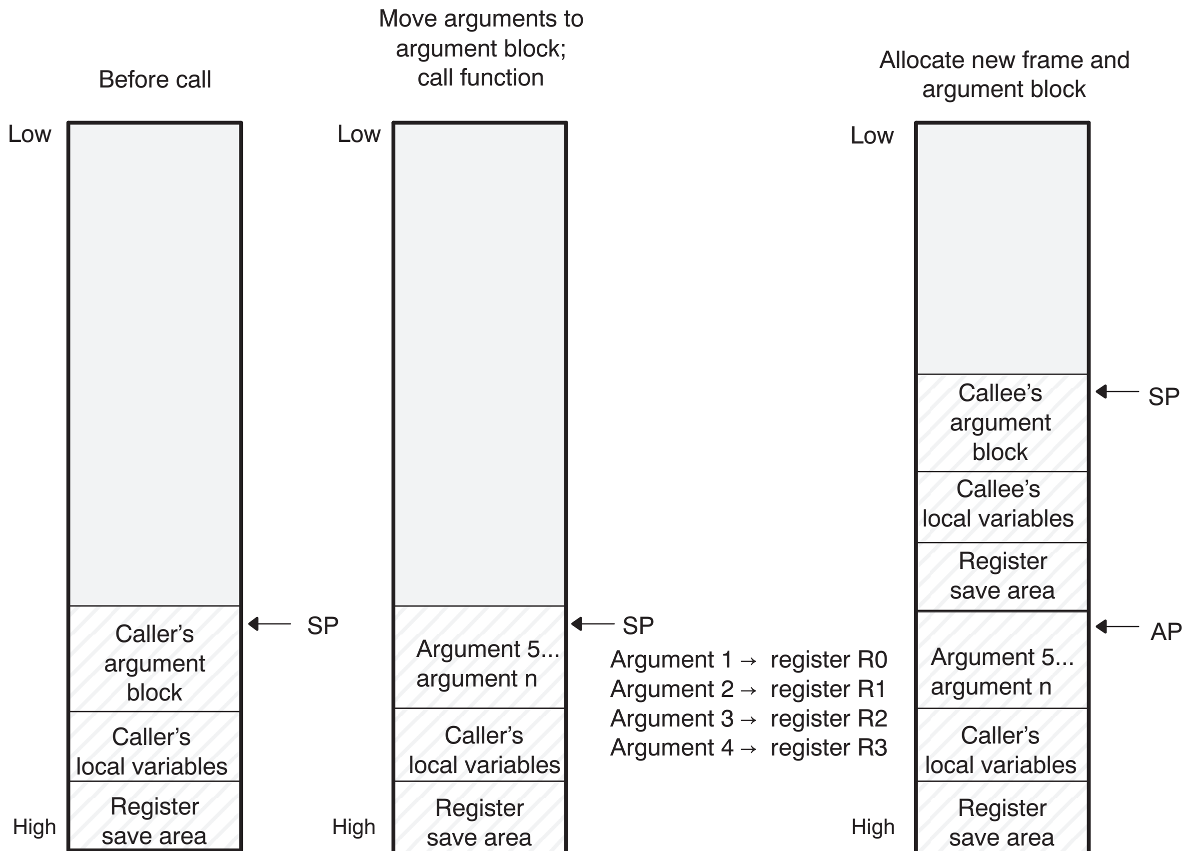
- **Argument block**. The part of the local frame used to pass arguments to other functions. Arguments are passed to a function by moving them into the argument block rather than pushing them on the stack. The local frame and argument block are allocated at the same time.
- **Register save area**. The 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 to V8 and LR). It is the called function's responsibility to preserve the 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 at http://www.arm.com/products/DevTools/ABI.html.

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.

Copyright © 2012, Texas Instruments Incorporated

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 6.10 for more information.)

Copyright © 2012, Texas Instruments Incorporated

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 6.6.
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.

Copyright © 2012, Texas Instruments Incorporated

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 (points to 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.

6.4.5 Generating Long Calls (-ml Option) in 16-bit Mode

You can use the -ml option when compiling C/C++ code to produce long calls in 16-bit mode. You must also use the -mt (16-bit mode) and -md (no dual state) options, or -ml (long call) has no effect.

The ARM C/C++ compiler normally uses the BL assembly instruction when making procedure calls in 16-bit mode. In object code, this is translated into two 16-bit BL instructions, each of which has 11 bits for the offset. Adding the two together, plus the assumed zero bit, gives only 23 bits of offset for a call. This is not enough when applications have widely disbursed areas of memory in which instructions reside.

The -ml option uses the BX instruction instead of the BL instruction. The BX instruction is less efficient. It requires a load instruction which uses a word of memory to load the destination address into a register, a move instruction to save the return address, and a BX instruction to make procedure calls.

6.5 Interfacing C and C++ With Assembly Language

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

- Use separate modules of assembled code and link them with compiled C/C++ modules (see Section 6.5.1).
- Use assembly language variables and constants in C/C++ source (see Section 6.5.2).
- Use inline assembly language embedded directly in the C/C++ source (see Section 6.5.4).
- Modify the assembly language code that the compiler produces (see Section 6.5.5).

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

Interfacing C/C++ with assembly language functions is straightforward if you follow the calling conventions defined in Section 6.4, and the register conventions defined in Section 6.3. C/C++ code can access variables and call functions defined in assembly language, and assembly code can access C/C++ variables and call C/C++ functions.

Follow these guidelines to interface assembly language and C:

- 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 6.6.
- When you call a C/C++ function from assembly language, load the designated registers with arguments and push the remaining arguments onto the stack as described in Section 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.

Copyright © 2012, Texas Instruments Incorporated

- 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 are writing assembly language code, you must use the same linknames as those assigned by the compiler. See Section 5.13 for more information.
- Any object or function declared in assembly language that is accessed or called from C/C++ must be declared with the .def or .global directive in the assembly language modifier. This declares the symbol as external and allows the linker to resolve references to it.

  Likewise, to access a C/C++ function or object from assembly language, declare the C/C++ object with the .ref or .global directive in the assembly language module. This creates an undeclared external reference that the linker resolves.

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.

Copyright © 2012, Texas Instruments Incorporated

6.5.2 Accessing Assembly Language Variables From C/C++

It is sometimes useful for a C/C++ program to access variables or constants defined in assembly language. There are several methods that you can use to accomplish this, depending on where and how the item is defined: a variable defined in the .bss section, a variable not defined in the .bss section, or a constant.

6.5.2.1 Accessing Assembly Language Global Variables

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

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

Example 6-4 and Example 6-3 show how you can access a variable defined in .bss.

Example 6-3. Assembly Language Variable Program

```
* Note the use of underscores in the following lines

       .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  */
```

6.5.2.2 Accessing Assembly Language Constants

You can define global constants in assembly language by using the .set, .def, and .global directives, or you can define them in a linker command file using a linker assignment statement. These constants are accessible from C/C++ only with the use of special operators.

For normal variables defined in C/C++ or assembly language, the symbol table contains the *address of the value* of the variable. For assembler constants, however, the symbol table contains the *value* of the constant. The compiler cannot tell which items in the symbol table are values and which are addresses.

If you try to access an assembler (or linker) constant by name, the compiler attempts to fetch a value from the address represented in the symbol table. To prevent this unwanted fetch, you must use the & (address of) operator to get the value. In other words, if x is an assembly language constant, its value in C/C++ is &x.

You can use casts and #defines to ease the use of these symbols in your program, as in Example 6-5 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 */
```


Copyright © 2012, Texas Instruments Incorporated

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

```
_table_size  .set    10000     ; define the constant
             .global _table_size  ; make it global
```

Because you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example 6-5, int is used. You can reference linker-defined symbols in a similar manner.

6.5.3 Sharing C/C++ Header Files With Assembly Source

You can use the .cdecls assembler directive to share C headers containing declarations and prototypes between C and assembly code. Any legal C/C++ can be used in a .cdecls block and the C/C++ declarations will cause suitable assembly to be generated automatically, allowing you to reference the C/C++ constructs in assembly code. For more information, see the C/C++ header files chapter in the *ARM Assembly Language Tools User's Guide*.

6.5.4 Using Inline Assembly Language

Within a C/C++ program, you can use the asm statement to insert a single line of assembly language into the assembly language file created by the compiler. A series of asm statements places sequential lines of assembly language into the compiler output with no intervening code. For more information, see Section 5.8.

The asm statement is useful for inserting comments in the compiler output. Simply start the assembly code string with a semicolon (;) as shown below:

```
asm(";*** this is an assembly language comment");
```

NOTE: Using the asm Statement

Keep the following in mind when using the asm statement:

- Be extremely careful not to disrupt the C/C++ environment. The compiler does not check or analyze the inserted instructions.
- Avoid inserting jumps or labels into C/C++ code because they can produce unpredictable results by confusing the register-tracking algorithms that the code generator uses.
- Do not change the value of a C/C++ variable when using an asm statement. This is because the compiler does not verify such statements. They are inserted as is into the assembly code, and potentially can cause problems if you are not sure of their effect.
- Do not use the asm statement to insert assembler directives that change the assembly environment.
- Avoid creating assembly macros in C code and compiling with the --symdebug:dwarf (or -g) option. The C environment's debug information and the assembly macro expansion are not compatible.

6.5.5 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.

Copyright © 2012, Texas Instruments Incorporated

6.6 Interrupt Handling

As long as you follow the guidelines in this section, you can interrupt and return to C/C++ code without disrupting the C/C++ environment. When the C/C++ environment is initialized, the startup routine does not enable or disable interrupts. If the system is initialized by way of a hardware reset, interrupts are disabled. If your system uses interrupts, you must handle any required enabling or masking of interrupts. Such operations have no effect on the C/C++ environment and are easily incorporated with asm statements or calling an assembly language function.

6.6.1 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 5.9.11.

6.6.2 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 5.9.11 and Section 5.5.2, respectively.

6.6.3 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.

Copyright © 2012, Texas Instruments Incorporated

6.6.4 How to Map Interrupt Routines to Interrupt Vectors

To map 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.

NOTE: Prefixing C/C++ Interrupt Routines

Remember, if you are using C/C++ interrupt routines, you must use the proper linkname. So for TI ARM9 ABI and TIABI, use _c_intIRQ instead of c_intIRQ, for example, for a function defined in C scope.

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
```


Copyright © 2012, Texas Instruments Incorporated

6.6.5 Using Software Interrupts

A software interrupt 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.

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 5.9.18.

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.

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.

6.6.6 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.
- 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 6.6.1 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 6.6.1.

Copyright © 2012, Texas Instruments Incorporated

6.7 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. The compiler uses a preset naming convention for each of these routines, as described in Section 6.7.1 for the TI ARM9 ABI and TIABI modes. Refer to http://www.arm.com/products/DevTools/ABI.html for information on the EABI mode.

6.7.1 Naming Conventions

For the TI ARM9 ABI and TIABI modes, the names of the intrinsic run-time-support arithmetic and conversion routines are made up of two parts:

- The *prefix* indicates the type of the routine and the state from which it can be called. Table 6-7 lists the possible prefixes.
- The *root* indicates the operation performed by the routine. Table 6-8 lists the possible roots.

The prefixes from Table 6-7 are combined with the roots from Table 6-8 to form the function names listed in Table 6-9.

Table 6-7. Naming Convention Prefixes

Prefix	Type of Operation	State Called From
FD$	Double-precision floating point	16-BIS
FD_	Double-precision floating point	32-BIS
FS$	Single-precision floating point	16_BIS
FS_	Single-precision floating point	32_BIS
I$	Signed integer	16_BIS
I_	Signed integer	32-BIS
U$	Unsigned integer	16-BIS
U_	Unsigned integer	32-BIS

Table 6-8. Summary of Run-Time-Support Arithmetic and Conversion Functions

		Type of Usage [(1)]			
Root	**Operation Performed**	**uint**	**int**	**single**	**double**
ADD	Addition			X	X
CMP	Comparison			X	X
DIV	Division	X	X	X	X
MOD	Modulus	X	X		
MUL	Multiplication			X	X
SUB	Subtraction			X	X
TOI	Conversion to signed integer			X	X
TOU	Conversion to unsigned integer			X	X
TOFS	Conversion to single-precision floating point	X	X		X
TOFD	Conversion to double-precision floating point	X	X	X	

(1) X indicates that the combination is supported
uint unsigned integer single single-precision floating point
int signed integer double double-precision floating point

Copyright © 2012, Texas Instruments Incorporated

The source files for these functions are in the rtsc.src and rtscpp.src libraries. The source code has comments that describe the operation of the functions. You can extract, inspect, and modify any of these functions; be sure you follow the register usage conventions summarized in Table 6-9.

Table 6-9. Run-Time-Support Function Register Usage Conventions

	Inputs			
Function	Operand1	Operand2	Output	Source Filename
FS$TOI	R0		R0	fs_toi16.asm
FS_TOI	R0		R0	fs_toi32.asm
FD$TOI	R0:R1		R0	fd_toi16.asm
FSD_TOI	R0:R1		R0	fd_toi32.asm
FS$TOU	R0		R0	fs_tou16.asm
FS_TOU	R0		R0	fs_tou32.asm
FD$TOU	R0:R1		R0	fd_tou16.asm
FD_TOU	R0:R1		R0	fd_tou32.asm
I$TOFS	R0		R0	i_tofs16.asm
I_TOFS	R0		R0	i_tofs32.asm
U$TOFS	R0		R0	u_tofs16.asm
U_TOFS	R0		R0	u_tofs32.asm
FD$TOFS	R0:R1		R0	fd_tos16.asm
FD_TOFS	R0:R1		R0	fs_tos32.asm
I$TOFD	R0		R0:R1	i_tofd16.asm
I_TOFD	R0		R0:R1	i_tofd32.asm
U$TOFD	R0		R0:R1	u_tofd16.asm
U_TOFD	R0		R0:R1	u_tofd32.asm
FS$TOFD	R0		R0:R1	fs_tofd16.asm
FS_TOFD	R0		R0:R1	fs_tofd32.asm
FS$ADD	R0	R1 [(1)]	R0	fs_add16.asm
FS_ADD	R0	R1 [(1)]	R0	fs_add32.asm
FD$ADD	R0:R1	R2:R3 [(1)]	R0:R1	fd_add16.asm
FD_ADD	R0:R1	R2:R3 [(1)]	R0:R1	fd_add32.asm
FS$SUV	R0	R1	R0	fs_add16.asm
FS_SUB	R0	R1	R0	fs_add32.asm
FD$SUB	R0:R1	R2:R3 [(1)]	R0:R1	fd_add`6.asm
FD_SUB	R0:R1	R2:R3 [(1)]	R0:R1	fd_add32.asm
FS$MUL	R0	R1 [(1)]	R0	fs_mul16.asm
FS_MUL	R0	R1 [(1)]	R0	fs_mul32.asm
FD$MUL	R0:R1	R2:R3 [(1)]	R0:R1	fd_mul16.asm
FD_MUL	R0:R1	R2:R3 [(1)]	R0:R1	fd_mul32.asm
FS$DIV	R0	R1	R0	fs_div16.asm
FS_DIV	R0	R1	R0	fs_div32.asm
FD$DIV	R0:R1	R2:R3 [(1)]	R0:R1	fd_div16.asm
FD_DIV	R0:R1	R2:R3 [(1)]	R0:R1	fs_div32.asm
I$DIV	R0	R1	R1	i_div16.asm
I_DIV	R0	R1	R1	i_div32.asm
U$DIV	R0	R1	R1	u_div16.asm
U_DIV	R0	R1	R1	u_div32.asm
I$MOD	R0	R1	R0	i_mod16.asm
I_MOD	R0	R1	R0	i_mod32.asm
U$MOD	R0	R1	R0	u_mod16.asm

(1) Value in the register is preserved.

Copyright © 2012, Texas Instruments Incorporated

Table 6-9. Run-Time-Support Function Register Usage Conventions (continued)

	Inputs			
Function	**Operand1**	**Operand2**	**Output**	**Source Filename**
U_MOD	R0	R1	R0	u_mod32.asm
FS$CMP	R0	R1 [(1)]	CPSR	fs_cmp16.asm
FS_CMP	R0	R1 [(1)]	CPSR	fs_omp32.asm
FD$CMP	R0:R1 [(1)]	R2:R3 [(1)]	CPSR	fd_cmp16.asm
FD_CMP	R0:R1 [(1)]	R2:R3 [(1)]	CPSR	fd_cmp32.asm

6.7.2 CPSR Register and Interrupt Intrinsics

The intrinsics in Table 6-10 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.

Table 6-10. CPSR and Interrupt C/C++ Compiler Intrinsics

C/C++ Compiler Intrinsic	Assembly Instruction	Description
void **_call_swi(**uint *src* **);**	**SWI $** *src*	Call a software interrupt. The *src* must be an immediate.
dst = uint **_disable_FIQ( ) ;**	**MRS** *dst* **, CPSR** **ORR** *tmp* **,** *dst* **, #0x40** **MSR CPSR ,** *tmp*	Disable FIQ interrupts and return previous CPSR value.
dst = uint **_disable_interrupts( ) ;**	**MRS** *dst* **, CPSR** **ORR** *tmp* **,** *dst* **, #0xc0** **MSR CPSR ,** *tmp*	Disable all interrupts and return previous CPSR value.
dst = uint **_disable_IRQ( ) ;**	**MRS** *dst* **, CPSR** **ORR** *tmp* **,** *dst* **, #0x80** **MSR CPSR ,** *tmp*	Disable IRQ interrupts and return previous CPSR value.
dst = uint **_enable_FIQ( ) ;**	**MRS** *dst* **, CPSR** **BIC** *tmp* **,** *dst* **, #0x40** **MSR CPSR ,** *tmp*	Enable FIQ interrupts and return previous CPSR value.
dst = uint **_enable_interrupts( ) ;**	**MRS** *dst* **, CPSR** **BIC** *tmp* **,** *dst* **, #0xc0** **MSR CPSR ,** *tmp*	Enable all interrupts and return previous CPSR value.
dst = uint **_enable_IRQ( ) ;**	**MRS** *dst* **, CPSR** **BIC** *tmp* **,** *dst* **, #0x80** **MSR CPSR ,** *tmp*	Enable IRQ interrupts and return previous CPSR value.
dst = uint **_get_CPSR( ) ;**	**MRS** *dst* **, CPSR**	Get the CPSR register.
void **_set_CPSR(**uint *src* **);** void **_restore_interupts(**uint *src* **);**	**MSR CPSR ,** *src*	Set the CPSR register.
void **_set_CPSR_flg(**uint *src* **);**	**MSR** *dst* **, CPSR**	Set the CPSR flag bits. The *src* is rotated by the intrinsic.

Copyright © 2012, Texas Instruments Incorporated

6.8 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);
```

6.9 System Initialization

Before you can run a C/C++ program, you must create the C/C++ run-time environment. The C/C++ boot routine performs this task using a function called c_int00 (or _c_int00). The run-time-support source library, rts.src, contains the source to this routine in a module named boot.c (or boot.asm).

To begin running the system, the c_int00 function can be called by reset hardware. You must link the c_int00 function with the other object files. This occurs automatically when you use the --rom_model or --ram_model link option and include a standard run-time-support library as one of the linker input files.

When C/C++ programs are linked, the linker sets the entry point value in the executable output file to the symbol c_int00.

The c_int00 function performs the following tasks to initialize the environment:

1. Switches to the appropriate mode, reserves space for the run-time stack, and sets up the initial value of the stack pointer (SP). In EABI mode, 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 6.9.2 for TI_ARM9_ABI and TIABI modes and Section 6.9.3 for EABI mode.
   - Calls C++ initialization routines for file scope construction from the global constructor table. For more information, see Section 6.9.3.6 for EABI mode and Section 6.9.2.3 for TI_ARM9_ABI and TIABI modes.
3. Calls the function main to run the C/C++ program.

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the operations listed above to correctly initialize the C/C++ environment.

Copyright © 2012, Texas Instruments Incorporated

6.9.1 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.

6.9.2 TI ARM9 ABI/TIABI Automatic Initialization of Variables

Some global variables must have initial values assigned to them before a C/C++ program starts running. The process of retrieving these variables' data and initializing the variables with the data is called autoinitialization.

The TIARM9/TIABI compiler builds tables in a special section called .cinit that contains data for initializing global and static variables. Each compiled module contains these initialization tables. The linker combines them into a single table (a single .cinit section). The boot routine or a loader uses this table to initialize all the system variables.

Initializing Variables

NOTE: In ANSI/ISO C, global and static variables that are not explicitly initialized must be set to 0 before program execution. The TIARM9/TIABI C/C++ compiler does not perform any preinitialization of uninitialized variables. Explicitly initialize any variable that must have an initial value of 0.

Global variables are either autoinitialized at run time or at load time; see Section 6.9.2.1 and Section 6.9.2.2. Also see Section 5.14.

6.9.2.1 Autoinitialization of Variables at Run Time

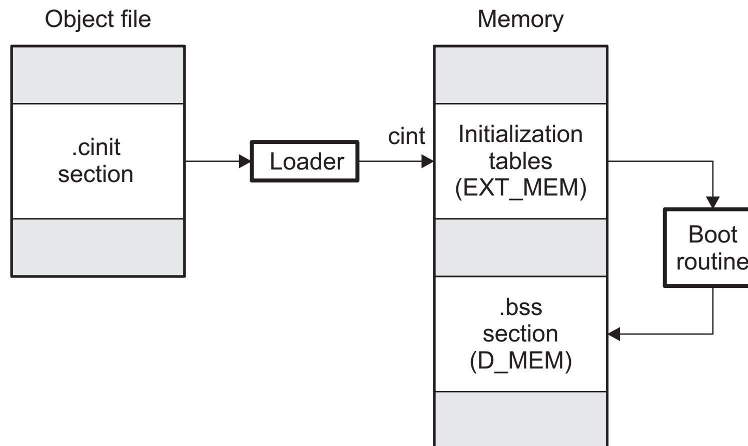
Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the --rom_model option.

Using this method, the .cinit section is loaded into memory along with all the other initialized sections, and global variables are initialized at run time. The linker defines a special symbol called cinit that points to the beginning of the initialization tables in memory. When the program begins running, the C/C++ boot routine copies data from the tables (pointed to by .cinit) into the specified variables in the .bss section. This allows initialization data to be stored in ROM and copied to RAM each time the program starts.

Figure 6-6 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

Copyright © 2012, Texas Instruments Incorporated

Figure 6-6. Autoinitialization at Run Time



6.9.2.2 Initialization of Variables at Load Time

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the --ram_model option.

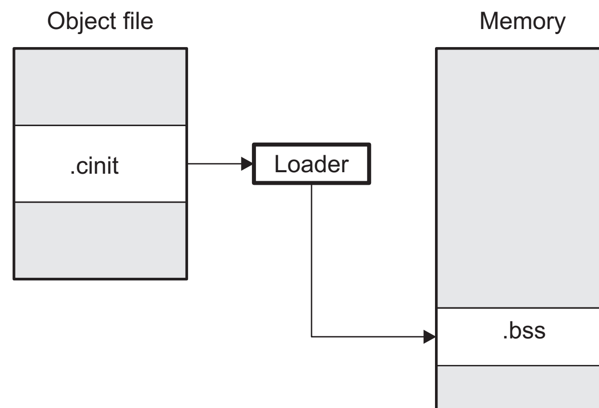
When you use the --ram_model link option, the linker sets the STYP_COPY bit in the .cinit section's header. This tells the loader not to load the .cinit section into memory. (The .cinit section occupies no space in the memory map.) The linker also sets the cinit symbol to -1 (normally, cinit points to the beginning of the initialization tables). This indicates to the boot routine that the initialization tables are not present in memory; accordingly, no run-time initialization is performed at boot time.

A loader (which is not part of the compiler package) must be able to perform the following tasks to use initialization at load time:

- Detect the presence of the .cinit section in the object file
- Determine that STYP_COPY is set in the .cinit section header, so that it knows not to copy the .cinit section into memory
- Understand the format of the initialization tables

Figure 6-7 illustrates the initialization of variables at load time.

Figure 6-7. Initialization at Load Time



Regardless of the use of the --rom_model or --ram_model options, the .pinit section is always loaded and processed at run time.

Copyright © 2012, Texas Instruments Incorporated

6.9.2.3 Global Constructors for COFF

All global C++ variables that have constructors must have their constructor called before main. The compiler builds a table in a section called .pinit of global constructor addresses that must be called, in order, before main. The linker combines the .pinit section form each input file to form a single table in the .pinit section. The boot routine uses this table to execute the constructors. (See Section 6.9.3.6 for global constructor details for EABI.)

6.9.3 EABI 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.

6.9.3.1 EABI 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++ EABI compiler supports preinitialization of uninitialized variables by default. This can be turned off by specifying the linker option --zero_init=off. TI ARM9 ABI does not support zero initialization.

6.9.3.2 EABI Direct Initialization

The EABI 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 6.9.3.5.

In the run-time initialization model, the linker uses the data in these sections to create initialization data and an additional initialization table. The boot routine processes the initialization table to copy data from load addresses to run addresses. See Section 6.9.3.3.

6.9.3.3 Autoinitialization of Variables at Run Time in EABI Mode

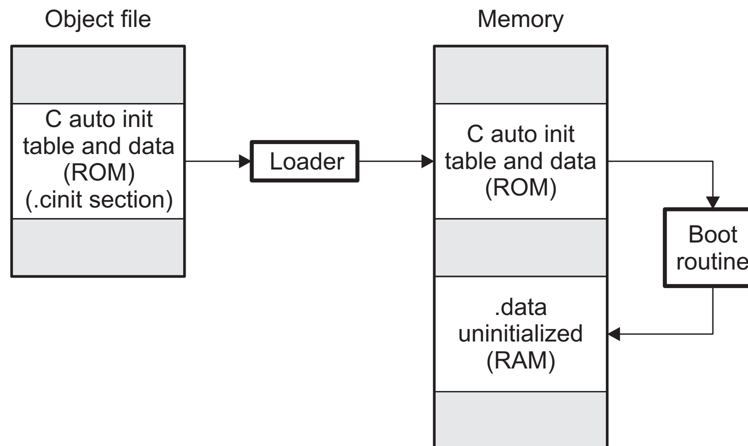
Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the --rom_model option.

Using this method, the linker creates an 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 6-8 illustrates autoinitialization at run time in EABI Mode. Use this method in any system where your application runs from code burned into ROM.

Copyright © 2012, Texas Instruments Incorporated

Figure 6-8. Autoinitialization at Run Time in EABI Mode



6.9.3.4 Autoinitialization Tables in EABI Mode

In EABI mode, 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.

Migration from COFF to ELF Initialization

NOTE: The name .cinit is used primarily to simplify migration from COFF to ELF format and the .cinit section created by the linker has nothing in common (except the name) with the COFF cinit records.

The autoinitialization table has the following format:

_TI_CINIT_Base:

32-bit load address	32-bit run address
⋮	⋮
32-bit load address	32-bit run address

_TI_CINT_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

Copyright © 2012, Texas Instruments Incorporated

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.

6.9.3.4.1 Length Followed by Data Format in EABI Mode

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.

6.9.3.4.2 Zero Initialization Format in EABI Mode

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.

Copyright © 2012, Texas Instruments Incorporated

6.9.3.4.3 Run Length Encoded (RLE) Format in EABI Mode

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).
         (i) If L == 0, the end of the data is reached, go to step 7.
         (ii) 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.

RLE Decompression Routine

NOTE: The previous decompression routine, __TI_decompress_rle(), is included in the run-time-support library for decompressing RLE encodings that are generated by older versions of the linker.

6.9.3.4.4 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.

6.9.3.4.5 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.

Copyright © 2012, Texas Instruments Incorporated

Example 6-8. Processing the C Autoinitialization Table

```
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);
   }
}
```

6.9.3.5 Initialization of Variables at Load Time in EABI Mode

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.

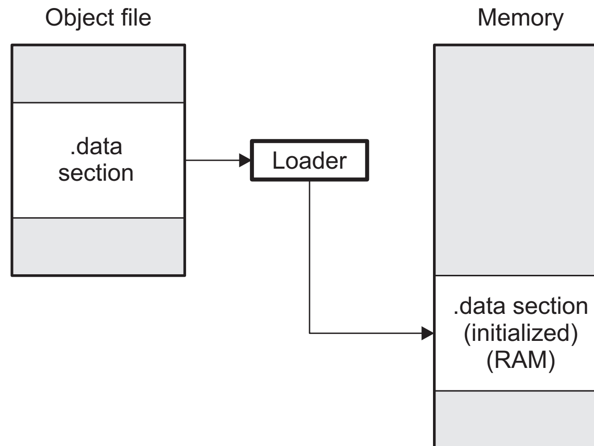
When you use the --ram_model link option, the linker does not generate C autoinitialization tables and data. The direct initialized sections (.data) in the compiled object files are combined according to the linker command file to generate initialized output sections. The loader loads the initialized 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.

Copyright © 2012, Texas Instruments Incorporated

Figure 6-9 illustrates the initialization of variables at load time in EABI mode.

Figure 6-9. Initialization at Load Time in EABI Mode



6.9.3.6 Global Constructors in EABI Mode

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 6-10. Constructor Table for EABI Mode

SHT$$INIT_ARRAY$$Base:

Address of constructor 1
Address of constructor 2
⋮
Address of constructor n

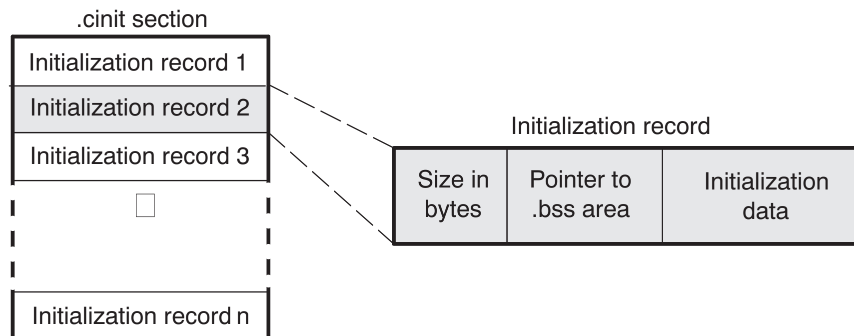
SHT$$INIT_ARRAY$$Limit:

6.9.4 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 6-11 shows the format of the .cinit section and the initialization records.

Copyright © 2012, Texas Instruments Incorporated

Figure 6-11. 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.

Example 6-9 shows initialized global variables defined in C. Example 6-10 shows the corresponding initialization table. The section .cinit:c is a subsection in the .cinit section that contains all scalar data. The subsection is handled as one record during initialization, which minimizes the overall size of the .cinit section.

Example 6-9. Initialized Variables Defined in C

```
int   i = 23;
int   a[5] = { 1, 2, 3, 4, 5 };
```


Copyright © 2012, Texas Instruments Incorporated

Example 6-10. Initialized Information for Variables Defined in Example 6-9

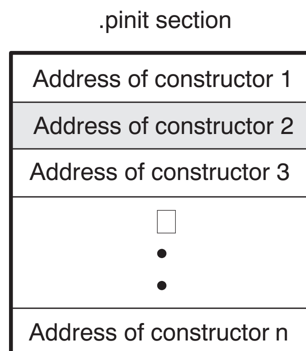
```
     .sect     ".cinit"          ; Initialization section
* Initialization record for variable i
     .align    4                 ; align on word boundary
     .field    4,32              ; length of data (1 word)
     .field    _i+0,32           ; address of i
     .field    23,32             ; _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 (see Figure 6-12). The constructors appear in the table after the .cinit initialization.

Figure 6-12. 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 5.5.1.

Copyright © 2012, Texas Instruments Incorporated

6.10 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 (see Section 5.11.2) 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.

6.10.1 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 6-11 to determine the best level of support to use for your code.

Table 6-11. 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 5.9.8 for more information.

Copyright © 2012, Texas Instruments Incorporated

- **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.

6.10.2 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.

6.10.2.1 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.

6.10.2.2 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 6-11 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 -mt 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 6-14 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 -mt option, creating 32-bit instructions.

Copyright © 2012, Texas Instruments Incorporated

Example 6-11. C Code Compiled for 16-BIS State: sum( )

```
int total = 0;

sum(int val1, int val2)
{
   int val = max(val1, val2);

   total += val;
}
```

Example 6-12. 16-Bit Assembly Program for Example 6-11

```
;***********************************************************
;* 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
```


Copyright © 2012, Texas Instruments Incorporated

Example 6-13. C Code Compiled for 32-BIS State: sum( )

```
int max(int a, int b)
{
   return a < b ? b : a;
}
```

Example 6-14. 32-Bit Assembly Program for Example 6-13

```
;************************************************************
;* 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( ).

Copyright © 2012, Texas Instruments Incorporated

Texas Instruments

Chapter 7
SPNU151H–August 2012

Using Run-Time-Support Functions and Building Libraries

Some of the features of C/C++ (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are provided as an ANSI/ISO C/C++ standard library, rather than as part of the compiler itself. The TI implementation of this library is the run-time-support library (RTS). The C/C++ compiler implements the complete ISO standard library except for those facilities that handle 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 inSection 7.1 and Section 7.2.

A library-build utility is provided with the code generation tools that lets you create customized run-time-support libraries. This process is described in Section 7.4 .

Topic **Page**

7.1 C and C++ Run-Time Support Libraries **156**
7.2 The C I/O Functions **159**
7.3 Handling Reentrancy (_register_lock() and _register_unlock() Functions) **171**
7.4 Library-Build Process **172**

Copyright © 2012, Texas Instruments Incorporated

7.1 C and C++ Run-Time Support Libraries

ARM compiler releases include pre-built run-time 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 7.4 for information on the library-naming conventions.

The run-time-support library contains the following:

- ANSI/ISO C/C++ standard library
- C I/O library
- Low-level support functions that provide I/O to the host operating system
- Fundamental arithmetic routines
- System startup routine, _c_int00
- Functions and macros that allow C/C++ to access specific instructions

The run-time-support libraries do not contain functions involving signals and locale issues.

The C++ library supports wide chars, in that template functions and classes that are defined for char are also available for wide char. For example, wide char stream classes wios, wiostream, wstreambuf and so on (corresponding to char classes ios, iostream, streambuf) are implemented. However, there is no low-level file I/O for wide chars. Also, the C library interface to wide char support (through the C++ headers <cwchar> and <cwctype>) is limited as described in Section 5.1.

The C++ library included with the compiler is licensed from Dinkumware, Ltd. The Dinkumware C++ library is a fully conforming, industry-leading implementation of the standard C++ library.

TI does not provide documentation that covers the functionality of the C++ library. TI suggests referring to one of the following sources:

- *The Standard C++ Library: A Tutorial and Reference*,Nicolai M. Josuttis, Addison-Wesley, ISBN 0-201-37926-0
- *The C++ Programming Language* (Third or Special Editions), Bjarne Stroustrup, Addison-Wesley, ISBN 0-201-88954-4 or 0-201-70073-5
- Dinkumware's online reference at http://dinkumware.com/manuals

7.1.1 Linking Code With the Object Library

When you link your program, you must specify the object library as one of the linker input files so that references to the I/O and run-time-support functions can be resolved. You can either specify the library or allow the compiler to select one for you. See Section 4.3.1 for further information.

When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see the *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.

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.

7.1.3 Modifying a Library Function

You can inspect or modify library functions by unzipping the source file (rtssrc.zip), changing the specific function file, and rebuilding the library. When extracted (with any standard unzip tool on windows, linux, or unix), this zip file recreates the run-time source tree for the run-time library.

The source for the libraries is included in the rtssrc.zip file. See Section 7.4 for details on rebuilding.

You can also build a new library this way, rather than rebuilding into rtsv4_A_be_eabi.lib. See Section 7.4.

Copyright © 2012, Texas Instruments Incorporated

7.1.4 Minimal Support for Internationalization

The library now includes the header files <locale.h>, <wchar.h>, and <wctype.h>, which provide APIs to support non-ASCII character sets and conventions. Our implementation of these APIs is limited in the following ways:

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

7.1.5 Allowable Number of Open Files

In the <stdio.h> header file, the value for the macro FOPEN_MAX has been changed from 12 to the value of the macro _NFILE, which is set to 10. The impact is that you can only have 10 files simultaneously open at one time (including the pre-defined streams - stdin, stdout, stderr).

The C standard requires that the minimum value for the FOPEN_MAX macro is 8. The macro determines the maximum number of files that can be opened at one time. The macro is defined in the stdio.h header file and can be modified by changing the value of the _NFILE macro.

7.1.6 Nonstandard Header Files in rtssrc.zip

The rtssrc.zip self-processing zip file 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.

Copyright © 2012, Texas Instruments Incorporated

7.1.7 Library Naming Conventions

By default, the linker uses automatic library selection to select the correct run-time-support library (see Section 4.3.1.1) for your application. If you select the library manually, you must select the matching library according to the following naming scheme:

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, 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, where *n* designates the version. Current values are:

- 2 VFPv2
- 3 VFPv3
- 3D16 VFPv3D16

abi Indicates the application binary interface (ABI) used:

- eabi
- tiarm7
- tiarm9

eh Indicates the library has exception handling support

Copyright © 2012, Texas Instruments Incorporated

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);
```

Debugger Required for Default HOST

NOTE: For the default HOST device to work, there must be a debugger to handle the C I/O requests; the default HOST device cannot work by itself in an embedded system. To work in an embedded system, you will need to provide an appropriate driver for your system.

NOTE: C I/O Mysteriously Fails

If there is not enough space on the heap for a C I/O buffer, operations on the file will silently fail. If a call to printf() mysteriously fails, this may be the reason. The heap needs to be at least large enough to allocate a block of size BUFSIZ (defined in stdio.h) for every file on which I/O is performed, including stdout, stdin, and stderr, plus allocations performed by the user's code, plus allocation bookkeeping overhead. Alternately, declare a char array of size BUFSIZ and pass it to setvbuf to avoid dynamic allocation. To set the heap size, use the --heap_size option when linking (refer to the *Linker Description* chapter in the*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 todal). The macro _NDEVICE controls how many device drivers are installed at one time (the HOST device counts against this total).

7.2.1 High-Level I/O Functions

The high-level functions are the standard C library of stream I/O routines (printf, scanf, fopen, getchar, and so on). These functions call one or more low-level I/O functions to carry out the high-level I/O request. The high-level I/O routines operate on FILE pointers, also called *streams*.

Portable applications should use only the high-level I/O functions.

To use the high-level I/O functions:

- Include the header file stdio.h 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 2-21.

Copyright © 2012, Texas Instruments Incorporated

For example, given the following C program in a file named main.c:

```
#include <stdio.h>;

void main()
{
   FILE *fid;

   fid = fopen("myfile","w");
   fprintf(fid,"Hello, world\n");
   fclose(fid);

   printf("Hello again, world\n");
}
```

Issuing the following compiler command compiles, links, and creates the file main.out from the run-time-support library:

```
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.

7.2.2 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.

Copyright © 2012, Texas Instruments Incorporated

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

Copyright © 2012, Texas Instruments Incorporated

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

Copyright © 2012, Texas Instruments Incorporated

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

Copyright © 2012, Texas Instruments Incorporated

rename — *Rename File*

Syntax for C

#include {<stdio.h> | <file.h>}

int rename (const char * *old_name* **, const char** * *new_name* **);**

Syntax for C++

#include {<cstdio> | <file.h>}

int std::rename (const char * *old_name* **, const char** * *new_name* **);**

Description

The rename function changes the name of a file.

- The *old_name* is the current name of the file.
- The *new_name* is the new name for the file.

> **NOTE:** The optional device specified in the new name must match the device of the old name. If they do not match, a file copy would be required to perform the rename, and rename is not capable of this action.

Return Value

The function returns one of the following values:

0	if successful
-1	on failure

> **NOTE:** Although rename is a low-level function, it is defined by the C standard and can be used by portable applications.

7.2.3 Device-Driver Level I/O Functions

At the next level are the device-level drivers. They map directly to the low-level I/O functions. The default device driver is the HOST device driver, which uses the debugger to perform file operations. The HOST device driver is automatically used for the default C streams stdin, stdout, and stderr.

The HOST device driver shares a special protocol with the debugger running on a host system so that the host can perform the C I/O requested by the program. Instructions for C I/O operations that the program wants to perform are encoded in a special buffer named _CIOBUF_ in the .cio section. The debugger halts the program at a special breakpoint (C$$IO$$), reads and decodes the target memory, and performs the requested operation. The result is encoded into _CIOBUF_, the program is resumed, and the target decodes the result.

The HOST device is implemented with seven functions, HOSTopen, HOSTclose, HOSTread, HOSTwrite, HOSTlseek, HOSTunlink, and HOSTrename, which perform the encoding. Each function is called from the low-level I/O function with a similar name.

A device driver is composed of seven required functions. Not all function need to be meaningful for all devices, but all seven must be defined. Here we show the names of all seven functions as starting with DEV, but you may chose any name except for HOST.

Copyright © 2012, Texas Instruments Incorporated

DEV_open — *Open File for I/O*

Syntax

int DEV_open (const char * *path* **, unsigned** *flags* **, int** *llv_fd* **);**

Description

This function finds a file matching *path* and opens it for I/O as requested by *flags*.

- The *path* is the filename of the file to be opened. If the name of a file passed to open has a device prefix, the device prefix will be stripped by open, so DEV_open will not see it. (See Section 7.2.5 for details on the device prefix.)
- The *flags* are attributes that specify how the file is manipulated. The flags are specified using the following symbols:

```
O_RDONLY   (0x0000)   /* open for reading */
O_WRONLY   (0x0001)   /* open for writing */
O_RDWR     (0x0002)   /* open for read & write */
O_APPEND   (0x0008)   /* append on each write */
O_CREAT    (0x0200)   /* open with file create */
O_TRUNC    (0x0400)   /* open with truncation */
O_BINARY   (0x8000)   /* open in binary mode */
```

  See POSIX for further explanation of the flags.
- The *llv_fd* is treated as a suggested low-level file descriptor. This is a historical artifact; newly-defined device drivers should ignore this argument. This differs from the low-level I/O open function.

This function must arrange for information to be saved for each file descriptor, typically including a file position indicator and any significant flags. For the HOST version, all the bookkeeping is handled by the debugger running on the host machine. If the device uses an internal buffer, the buffer can be created when a file is opened, or the buffer can be created during a read or write.

Return Value

This function must return -1 to indicate an error if for some reason the file could not be opened; such as the file does not exist, could not be created, or there are too many files open. The value of errno may optionally be set to indicate the exact error (the HOST device does not set errno). Some devices might have special failure conditions; for instance, if a device is read-only, a file cannot be opened O_WRONLY.

On success, this function must return a non-negative file descriptor unique among all open files handled by the specific device. It need not be unique across devices. Only the low-level I/O functions will see this device file descriptor; the low-level function open will assign its own unique file descriptor.

Copyright © 2012, Texas Instruments Incorporated

DEV_close — *Close File for I/O*

Syntax

int DEV_close (int *dev_fd* **);**

Description

This function closes a valid open file descriptor.

On some devices, DEV_close may need to be responsible for checking if this is the last file descriptor pointing to a file that was unlinked. If so, it is responsible for ensuring that the file is actually removed from the device and the resources reclaimed, if appropriate.

Return Value

This function should return -1 to indicate an error if the file descriptor is invalid in some way, such as being out of range or already closed, but this is not required. The user should not call close() with an invalid file descriptor.

DEV_read — *Read Characters from a File*

Syntax

int DEV_read (int *dev_fd* **, char *** *bu* **, unsigned** *count* **);**

Description

The read function reads *count* bytes from the input file associated with *dev_fd*.

- The *dev_fd* is the number assigned by open to an opened file.
- The *buf* is where the read characters are placed.
- The *count* is the number of characters to read from the file.

Return Value

This function must return -1 to indicate an error if for some reason no bytes could be read from the file. This could be because of an attempt to read from a O_WRONLY file, or for device-specific reasons.

If count is 0, no bytes are read and this function returns 0.

This function returns the number of bytes read, from 0 to count. 0 indicates that EOF was reached before any bytes were read. It is not an error to read less than count bytes; this is common if the are not enough bytes left in the file or the request was larger than an internal device buffer size.

DEV_write — *Write Characters to a File*

Syntax

int DEV_write (int *dev_fd* **, const char *** *buf* **, unsigned** *count* **);**

Description

This function writes *count* bytes to the output file.

- The *dev_fd* is the number assigned by open to an opened file.
- The *buffer* is where the write characters are placed.
- The *count* is the number of characters to write to the file.

Return Value

This function must return -1 to indicate an error if for some reason no bytes could be written to the file. This could be because of an attempt to read from a O_RDONLY file, or for device-specific reasons.

Copyright © 2012, Texas Instruments Incorporated

DEV_lseek — *Set File Position Indicator*

Syntax **off_t lseek (int** *dev_fd* **, off_t** *offset* **, int** *origin* **);**

Description This function sets the file's position indicator for this file descriptor as lseek.

If lseek is supported, it should not allow a seek to before the beginning of the file, but it should support seeking past the end of the file. Such seeks do not change the size of the file, but if it is followed by a write, the file size will increase.

Return Value If successful, this function returns the new value of the file position indicator.

This function must return -1 to indicate an error if for some reason no bytes could be written to the file. For many devices, the lseek operation is nonsensical (e.g. a computer monitor).

DEV_unlink — *Delete File*

Syntax **int DEV_unlink (const char** * *path* **);**

Description Remove the association of the pathname with the file. This means that the file may no longer by opened using this name, but the file may not actually be immediately removed.

Depending on the device, the file may be immediately removed, but for a device which allows open file descriptors to point to unlinked files, the file will not actually be deleted until the last file descriptor is closed. See Section 7.2.3.

Return Value This function must return -1 to indicate an error if for some reason the file could not be unlinked (delayed removal does not count as a failure to unlink.)

If successful, this function returns 0.

DEV_rename — *Rename File*

Syntax **int DEV_rename (const char** * *old_name* **, const char** * *new_name* **);**

Description This function changes the name associated with the file.

- The *old_name* is the current name of the file.
- The *new_name* is the new name for the file.

Return Value This function must return -1 to indicate an error if for some reason the file could not be renamed, such as the file doesn't exist, or the new name already exists.

NOTE: It is inadvisable to allow renaming a file so that it is on a different device. In general this would require a whole file copy, which may be more expensive than you expect.

If successful, this function returns 0.

Copyright © 2012, Texas Instruments Incorporated

7.2.4 Adding a User-Defined Device Driver for C I/O

The function add_device allows you to add and use a device. When a device is registered with add_device, the high-level I/O routines can be used for I/O on that device.

You can use a different protocol to communicate with any desired device and install that protocol using add_device; however, the HOST functions should not be modified. The default streams stdin, stdout, and stderr can be remapped to a file on a user-defined device instead of HOST by using freopen() as in Example 7-1. If the default streams are reopened in this way, the buffering mode will change to _IOFBF (fully buffered). To restore the default buffering behavior, call setvbuf on each reopened file with the appropriate value (_IOLBF for stdin and stdout, _IONBF for stderr).

The default streams stdin, stdout, and stderr can be mapped to a file on a user-defined device instead of HOST by using freopen() as shown in Example 7-1. Each function must set up and maintain its own data structures as needed. Some function definitions perform no action and should just return.

Example 7-1. Mapping Default Streams to Device

```
#include <stdio.h>
#include <file.h>
#include "mydevice.h"

void main()
{
    add_device("mydevice", _MSA,
               MYDEVICE_open, MYDEVICE_close,
               MYDEVICE_read, MYDEVICE_write,
               MYDEVICE_lseek, MYDEVICE_unlink, MYDEVICE_rename);

    /*-----------------------------------------------------------------------*/
    /* Re-open stderr as a MYDEVICE file                                     */
    /*-----------------------------------------------------------------------*/
    if (!freopen("mydevice:stderrfile", "w", stderr))
    {
        puts("Failed to freopen stderr");
        exit(EXIT_FAILURE);
    }

    /*-----------------------------------------------------------------------*/
    /* stderr should not be fully buffered; we want errors to be seen as     */
    /* soon as possible.  Normally stderr is line-buffered, but this example */
    /* doesn't buffer stderr at all.  This means that there will be one call */
    /* to write() for each character in the message.                         */
    /*-----------------------------------------------------------------------*/
    if (setvbuf(stderr, NULL, _IONBF, 0))
    {
        puts("Failed to setvbuf stderr");
        exit(EXIT_FAILURE);
    }

    /*-----------------------------------------------------------------------*/
    /* Try it out!                                                           */
    /*-----------------------------------------------------------------------*/
    printf("This goes to stdout\n");
    fprintf(stderr, "This goes to stderr\n"); }
```

NOTE: Use Unique Function Names

The function names open, read, write, close, lseek, rename, and unlink are used by the low-level routines. Use other names for the device-level functions that you write.

Copyright © 2012, Texas Instruments Incorporated

Use the low-level function add_device() to add your device to the device_table. The device table is a statically defined array that supports *n* devices, where *n* is defined by the macro _NDEVICE found in stdio.h/cstdio.

The first entry in the device table is predefined to be the host device on which the debugger is running. The low-level routine add_device() finds the first empty position in the device table and initializes the device fields with the passed-in arguments. For a complete description, see the add_device function.

7.2.5 The device Prefix

A file can be opened to a user-defined device driver by using a device prefix in the pathname. The device prefix is the device name used in the call to add_device followed by a colon. For example:

```
FILE *fptr = fopen("mydevice:file1", "r");
int fd = open("mydevice:file2, O_RDONLY, 0);
```

If no device prefix is used, the HOST device will be used to open the file.

add_device — *Add Device to Device Table*

Syntax for C

#include <file.h>

int add_device(char * *name*,
unsigned *flags* **,**
int (* *dopen* **)(const char ***path*, **unsigned** *flags*, **int llv_fd),**
int (* *dclose* **)(int dev_fd),**
int (* *dread* **)(int***dev_fd*, **char ***buf*, **unsigned** *count*),
int (* *dwrite* **)(int** *dev_fd*, **const char ***buf*, **unsigned** *count*),
off_t (* *dlseek* **)(int dev_fd, off_t** *ioffset*, **int** *origin*),
int (* *dunlink* **)(const char *** *path*),
int (* *drename* **)(const char ***old_name*, **const char ***new_name*));

Defined in

lowlev.c in rtssrc.zip

Description

The add_device function adds a device record to the device table allowing that device to be used for I/O from C. The first entry in the device table is predefined to be the HOST device on which the debugger is running. The function add_device() finds the first empty position in the device table and initializes the fields of the structure that represent a device.

To open a stream on a newly added device use fopen( ) with a string of the format *devicename* **:** *filename* as the first argument.

- The *name* is a character string denoting the device name. The name is limited to 8 characters.
- The *flags* are device characteristics. The flags are as follows:

  _SSA Denotes that the device supports only one open stream at a time

  _MSA Denotes that the device supports multiple open streams

  More flags can be added by defining them in file.h.
- The *dopen*, *dclose*, *dread*, *dwrite*, *dlseek*, *dunlink*, and *drename* specifiers are function pointers to the functions in the device driver that are called by the low-level functions to perform I/O on the specified device. You must declare these functions with the interface specified in Section 7.2.2. The device driver for the HOST that the 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

Copyright © 2012, Texas Instruments Incorporated

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);
}
```


Copyright © 2012, Texas Instruments Incorporated

7.3 Handling Reentrancy (_register_lock() and _register_unlock() Functions)

The C standard assumes only one thread of execution, with the only exception being extremely narrow support for signal handlers. The issue of reentrancy is avoided by not allowing you to do much of anything in a signal handler. However, BIOS applications have multiple threads which need to modify the same global program state, such as the CIO buffer, so reentrancy is a concern.

Part of the problem of reentrancy remains your responsibility, but the run-time-support environment does provide rudimentary support for multi-threaded reentrancy by providing support for critical sections. This implementation does not protect you from reentrancy issues such as calling run-time-support functions from inside interrupts; this remains your responsibility.

The run-time-support environment provides hooks to install critical section primitives. By default, a single-threaded model is assumed, and the critical section primitives are not employed. In a multi-threaded system such as BIOS, the kernel arranges to install semaphore lock primitive functions in these hooks, which are then called when the run-time-support enters code that needs to be protected by a critical section.

Throughout the run-time-support environment where a global state is accessed, and thus needs to be protected with a critical section, there are calls to the function _lock(). This calls the provided primitive, if installed, and acquires the semaphore before proceeding. Once the critical section is finished, _unlock() is called to release the semaphore.

Usually BIOS is responsible for creating and installing the primitives, so you do not need to take any action. However, this mechanism can be used in multi-threaded applications which do not use the BIOS LCK mechanism.

You should not define the functions _lock() and _unlock() functions directly; instead, the installation functions are called to instruct the run-time-support environment to use these new primitives:

```
void _register_lock  (void (  *lock)());

void _register_unlock(void (*unlock)());
```

The arguments to _register_lock() and _register_unlock() should be functions which take no arguments and return no values, and which implement some sort of global semaphore locking:

```
extern volatile sig_atomic_t *sema = SHARED_SEMAPHORE_LOCATION;
static int sema_depth = 0;
static void my_lock(void)
{
    while (ATOMIC_TEST_AND_SET(sema, MY_UNIQUE_ID) != MY_UNIQUE_ID);
    sema_depth++;
}
static void my_unlock(void)
{
    if (!--sema_depth) ATOMIC_CLEAR(sema);
}
```

The run-time-support nests calls to _lock(), so the primitives must keep track of the nesting level.

Copyright © 2012, Texas Instruments Incorporated

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)
- unzip (InfoZIP unzip 5.51 or later, or equivalent)

  You can download the software from http://www.info-zip.org.
- gmake (GNU make 3.81 or later)

  More information is available from GNU at http://www.gnu.org/software/make. GNU make (gmake) is also available in earlier versions of Code Composer Studio. GNU make is also included in some Unix support packages for Windows, such as the MKS Toolkit, Cygwin, and Interix. The GNU make used on Windows platfortms should expliclty report "This program build for Windows32" when the following is executed from the Command Prompt window:

```
gmake -h
```

All three of these programs are provided as a non-optional feature of CCS 5.1. They are also available as part of the optional XDC Tools feature if you are using an earlier version of CCS.

The mklib program looks for these executables in the following order:

1. in your PATH
2. in the directory getenv("CCS_UTILS_DIR")/cygwin
3. in the directory getenv("CCS_UTILS_DIR")/bin
4. in the directory getenv("XDCROOT")
5. in the directory getenv("XDCROOT")/bin

If you are invoking mklib from the command line, and these executables are not in your path, you must set the environment variable CCS_UTILS_DIR such that getenv("CCS_UTILS_DIR")/bin contains the correct programs.

7.4.2 Using the Library-Build Process

You should normally let the linker automatically rebuild libraries as needed. If necessary, you can run mklib directly to populate libraries. See Section 7.4.2.2 for situations when you might want to do this.

7.4.2.1 Automatic Standard Library Rebuilding by the Linker

The linker looks for run-time-support libraries primarily through the 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 is explicitly named (e.g. rtsv4_A_be_eabi), run-time support looks for that library exactly; otherwise, it uses the index library libc.a to pick an appropriate library.

The index library describes a set of libraries with different build attributes. The linker will compare the build attributes for each potential library with the build attributes of the application and will pick the best fit. For details on the index library, see the archiver chapter in the *ARM Assembly Language Tools User's Guide*.

Copyright © 2012, Texas Instruments Incorporated

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 will will invoke mklib to build it. This happens when the library is missing, regardless of whether the user specified the name of the library directly or allowed the linker to pick the best library from the index library.

The mklib program builds the requested library and places it in 'lib' directory part of 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.

Copyright © 2012, Texas Instruments Incorporated

7.4.2.2.3 Building Libraries With Custom Options

You can build a library with any extra custom options desired. This is useful for building a debugging version of the library, or with silicon exception workarounds enabled. The generated library is not a standard library, and must not be placed in the 'lib' directory. It should be placed in a directory local to the project which needs it. To build a debugging version of the library 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 7-1.

```
mklib --help
```

Table 7-1. The mklib Program Options

Option	Effect
--index=*filename*	The index library (libc.a) for this release. Used to find a template library for custom builds, and to find the source files (rtssrc.zip). REQUIRED.
--pattern=*filename*	Pattern for building a library. If neither --extra_options nor --options are specified, the library will be the standard library with the standard options for that library. If either --extra_options or --options are specified, the library is a custom library with custom options. REQUIRED unless --all is used.
--all	Build all standard libraries at once.
--install_to=*directory*	The directory into which to write the library. For a standard library, this defaults to the same directory as the index library (libc.a). For a custom library, this option is REQUIRED.
--compiler_bin_dir= *directory*	The directory where the compiler executables are. When invoking mklib directly, the executables should be in the path, but if they are not, this option must be used to tell mklib where they are. This option is primarily for use when mklib is invoked by the linker.
--name=*filename*	File name for the library with no directory part. Only useful for custom libraries.
--options='*str***'**	Options to use when building the library. The default options (see below) are *replaced* by this string. If this option is used, the library will be a custom library.
--extra_options='*str***'**	Options to use when building the library. The default options (see below) are also used. If this option is used, the library will be a custom library.
--list_libraries	List the libraries this script is capable of building and exit. ordinary system-specific directory.
--log=*filename*	Save the build log as *filename*.
--tmpdir=*directory*	Use *directory* for scratch space instead of the ordinary system-specific directory.
--gmake=*filename*	Gmake-compatible program to invoke instead of "gmake"
--parallel=*N*	Compile *N* files at once ("gmake -j N").
--query=*filename*	Does this script know how to build FILENAME?
--help or **--h**	Display this help.
--quiet or **--q**	Operate silently.
--verbose or **--v**	Extra information to debug this executable.

Examples:

To build all standard libraries and place them in the compiler's library directory:

```
mklib --all --index=$C_DIR/lib
```

To build one standard library and place it in the compiler's library directory:

```
mklib --pattern=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
```


Copyright © 2012, Texas Instruments Incorporated

7.4.3 Extending mklib

The **mklib** API is a uniform interface that allows Code Composer Studio to build libraries without needing to know exactly what underlying mechanism is used to build it. Each library vendor (e.g. the TI compiler) provides a library-specific copy of 'mklib' in the library directory that can be invoked, which understands a standardized set of options, and understands how to build the library. This allows the linker to automatically build application-compatible versions of any vendor's library without needing to register the library in advance, as long as the vendor supports mklib.

7.4.3.1 Underlying Mechanism

The underlying mechanism can be anything the vendor desires. For the compiler run-time-support libraries, mklib is just a wrapper which knows how to unpack Makefile from rtssrc.zip and invoke gmake with the appropriate options to build each library. If necessary, mklib can be bypassed and Makefile used directly, but this mode of operation is not supported by TI, and the you are responsible for any changes to Makefile. The format of the Makefile and the interface between mklib and the Makefile is subject to change without notice. The mklib program is the forward-compatible path.

7.4.3.2 Libraries From Other Vendors

Any vendor who wishes to distribute a library that can be rebuilt automatically by the linker must provide:

- An index library (like 'libc.a', but with a different name)
- A copy of mklib specific to that library
- A copy of the library source code (in whatever format is convenient)

These things must be placed together in one directory that is part of the linker's library search path (specified either in 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 7-1 without error, even if they do not do anything.

Copyright © 2012, Texas Instruments Incorporated

Chapter 8
SPNU151H–August 2012

C++ Name Demangler

The C++ compiler implements function overloading, operator overloading, and type-safe linking by encoding a function's prototype and namespace in its link-level name. The process of encoding the prototype into the linkname is often referred to as name mangling. When you inspect mangled names, such as in assembly files, disassembler output, or compiler or linker diagnostics, it can be difficult to associate a mangled name with its corresponding name in the C++ source code. The C++ name demangler is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code.

These topics tell you how to invoke and use the C++ name demangler. The C++ name demangler reads in input, looking for mangled names. All unmangled text is copied to output unaltered. All mangled names are demangled before being copied to output.

Topic **Page**

8.1 Invoking the C++ Name Demangler 177
8.2 C++ Name Demangler Options 177
8.3 Sample Usage of the C++ Name Demangler 178

Copyright © 2012, Texas Instruments Incorporated

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. (Options are discussed in Section 8.2.)
filenames	Text input files, such as the assembly file output by the compiler, the assembler listing file, the disassembly file, and the linker map file. If no filenames are specified on the command line, 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.

8.2 C++ Name Demangler Options

The following options apply only to the C++ name demangler:

--abi=eabi	Demangles EABI identifiers
-h	Prints a help screen that provides an online summary of the C++ name demangler options
-o *file*	Outputs to the given file rather than to standard out
-u	Specifies that external names do not have a C++ prefix
-v	Enables verbose mode (outputs a banner)

Copyright © 2012, Texas Instruments Incorporated

8.3 Sample Usage of the C++ Name Demangler

The examples in this section illustrate the demangling process. Example 8-1 shows a sample C++ program. Example 8-2 shows the resulting assembly that is output by the compiler. In this example, the linknames of all the functions are mangled; that is, their signature information is encoded into their names.

Example 8-1. C++ Code for calories_in_a_banana

```
class banana {

public:

    int calories(void);

    banana();

    ~banana();

};

int calories_in_a_banana(void)
{
    banana x;

    return x.calories();
}
```

Example 8-2. Resulting Assembly for calories_in_a_banana

```
calories_in_a_banana__Fv:
;* ----------------------------------------------------------------------------*
        SUB.W      #4,SP
        MOV.W      SP,r12                  ; |10|
        ADD.W      #2,r12                  ; |10|
        CALL       #__ct__6bananaFv        ; |10|
                                             ; |10|
        MOV.W      SP,r12                  ; |11|
        ADD.W      #2,r12                  ; |11|
        CALL       #calories__6bananaFv    ; |11|
                                             ; |11|
        MOV.W      r12,0(SP)               ; |11|
        MOV.W      SP,r12                  ; |11|
        ADD.W      #2,r12                  ; |11|
        MOV.W      #2,r13                  ; |11|
        CALL       #__dt__6bananaFv        ; |11|
                                             ; |11|
        MOV.W      0(SP),r12               ; |11|
        ADD.W      #4,SP
        RET
```

Executing the C++ name demangler demangles all names that it believes to be mangled. Enter:

```
armdem calories_in_a_banana.asm
```

The result is shown in Example 8-3. The linknames in Example 8-2 ___ct__6bananaFv, _calories__6bananaFv, and ___dt__6bananaFv are demangled.

Copyright © 2012, Texas Instruments Incorporated

Example 8-3. Result After Running the C++ Name Demangler

```
calories_in_a_banana():
;* --------------------------------------------------------------------------*
        SUB.W     #4,SP
        MOV.W     SP,r12                ; |10|
        ADD.W     #2,r12                ; |10|
        CALL      #banana::banana()      ; |10|
                                          ; |10|
        MOV.W     SP,r12                ; |11|
        ADD.W     #2,r12                ; |11|
        CALL      #banana::calories()  ; |11|
                                          ; |11|
        MOV.W     r12,0(SP)             ; |11|
        MOV.W     SP,r12                ; |11|
        ADD.W     #2,r12                ; |11|
        MOV.W     #2,r13                ; |11|
        CALL      #banana::~banana()      ; |11|
                                          ; |11|
        MOV.W     0(SP),r12             ; |11|
        ADD.W     #4,SP
        RET
```


Copyright © 2012, Texas Instruments Incorporated

Static Stack Depth Profiler

The static stack depth profiler provides information about the maximum stack depth requirements of an application based on the static information available in the linker output file. The application must be compiled with DWARF (default) type debug information under the TIABI (COFF) mode.

The profiler is a stand-alone application called armsdp. The profiler takes a linked output file as input and produces a listing that details the stack usage of all of the functions defined in the application. If an application contains indirect calls and/or reentrant procedures, then a configuration file should also be provided as input to the profiler.

Topic **Page**

9.1 Invoking the Static Stack Depth Profiler **181**
9.2 Stack Depth Statistics Listing **181**
9.3 Dependencies and Limitations **183**
9.4 User Input Mechanisms **183**
9.5 Configuration File Specification **184**
9.6 Profiler Generated Warnings **187**
9.7 Run-Time Support for Stack Depth Profiling **187**

Copyright © 2012, Texas Instruments Incorporated

9.1 Invoking the Static Stack Depth Profiler

The syntax for invoking the static stack depth profiler is as follows:

armsdp [**-c** *config*] *outfile*

armsdp	Command that runs the compiler and the assembler
-c *config*	Identifies a configuration file to be used by the profiler to supply information about indirectly called functions and reentrant procedures. For more information, see Section 9.5.
outfile	Identifies the linked output file for an application to be analyzed by the profiler. This file contains debug information about all functions included in the final link of an application.

The entry point, or root function, is identified in the linked output file. The profiler uses the same method to determine the entry point as the linker:

1. A global symbol identified with -e linker command option
2. The value of the _c_int00 symbol (if present)
3. The value of the _main or $main symbol (if present)
4. Address 0x0 (the default)

9.2 Stack Depth Statistics Listing

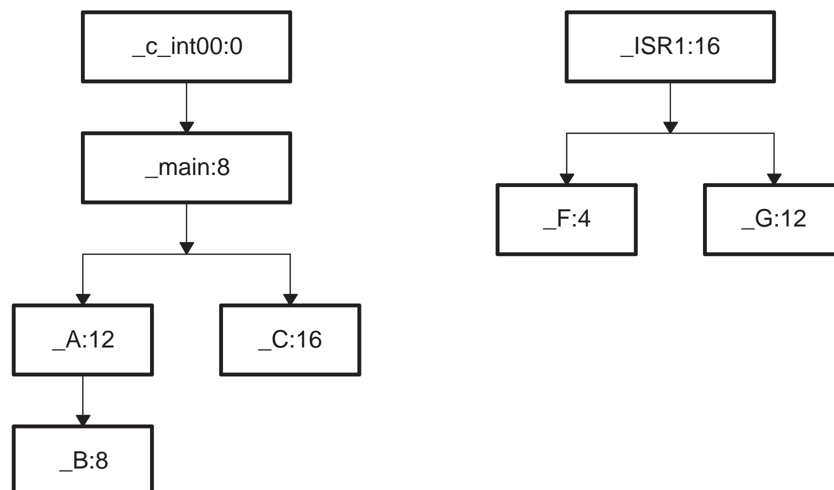
Each segment of the listing details the stack usage for a particular function in the application. Function segments are listed in order, beginning with the function that has the highest total stack need.

The first line in a segment provides details about a given parent function. The first line in a function segment provides information on the stack space needed by the function and its total stack usage estimate (the function's stack usage plus the worst stack usage of its callees).

Subsequent lines in a function segment list the callees of the parent function listed in the first line of the segment. The callees are listed in order from highest to lowest stack usage.

Figure 9-1 illustrates the application call trees for an example application:

Figure 9-1. Application Call Trees



Example 9-1 shows the function segment listings for the application illustrated in Figure 9-1.

Copyright © 2012, Texas Instruments Incorporated

Example 9-1. Profile of the Application Call Trees Figure

```
********************************************************************
* Static Stack Depth Analysis Profile
********************************************************************
FCN:cint00     stack usage:  0, total stack need:      28
    _main        stack usage:  8, subtree stack need:  28
====================================================================
FCN:_main      stack usage:  8, total stack need:      28
   _A            stack usage: 12, subtree stack need:  20
   _C            stack usage: 16, subtree stack need:  16
====================================================================
FCN:_ISR1      stack usage: 16, total stack need:      28
   _G            stack usage: 12, subtree stack need:  12
   _F            stack usage:  4, subtree stack need:   4
====================================================================
FCN:_A         stack usage: 12, total stack need:      20
   _B            stack usage:  8, subtree stack need:   8
====================================================================
FCN:_C         stack usage: 16, total stack need:      16
====================================================================
FCN:_G         stack usage: 12, total stack need:      12
====================================================================
FCN:_B         stack usage:  8, total stack need:       8
====================================================================
FCN:_F         stack usage:  4, total stack need:       4
====================================================================
```

A summary of the application's stack depth requirements is provided at the end of the listing. The summary states an estimate of the stack depth usage from each function in the application that does not have a parent. You can derive a worst case estimate of an application's stack depth using the information provided in the listing in combination with your knowledge of which functions are interrupts (and which stack mode those interrupts assume).

For example, a summary of Example 9-1 looks like this:

```
================ROOT FUNCTIONS==============================
_c_int00      total stack need:      28 bytes
_ISR1         total stack need:      28 bytes
```


Copyright © 2012, Texas Instruments Incorporated

9.3 Dependencies and Limitations

The profiler constructs a call tree based on the DWARF debug information provided in the linker output file. The tree is annotated with the stack usage information that the profiler derives from the function's debug information.

There are several significant limitations to the profiler's ability to complete this information:

- Hand-coded assembly functions and stack usage information for those functions
- Indirectly called functions and their callers
- Reentrant functions and their depth of recursion

The profiler relies on your input to supply this information in the application source code and in the configuration file (specified with the -c option). The assembler processes the assembly source code annotations (.asmfunc/.endasmfunc directives) and encodes the information into the linked output file in a form that the profiler can understand. You must also provide a configuration file to identify indirect calls and reentrant procedures to the profiler. The profiler uses this configuration file to annotate the call graph that it constructed from the linked output file for the application.

The accuracy of the stack depth estimate is heavily dependent on the accuracy of the information you provide. You must actively maintain this information throughout the life of a given application.

9.4 User Input Mechanisms

The assembler supports the .asmfunc and .endasmfunc assembler directives for identifying the beginning and end of an assembly function. The .asmfunc directive has an optional parameter for specifying the stack space needed for an assembly function.

For example, you could write a function, $foo(), which uses 12 bytes of stack space, as shown in Example 9-2:

Example 9-2. An Assembly Function

```
        .global      $foo
$foo:   .asmfunc     stack_usage(12)
        PUSH         { R4, R5, R6 }
        []
        POP          { R4, R5, R6 }
        MOV          PC, LR
        .endasmfunc
```

The .asmfunc and .endasmfunc directives identify the entry and exit points of the function. This information and the fact that $foo uses 12 bytes of stack space are then encoded into the debug information. Assembler functions must be annotated with these directives to be considered for stack depth analysis and profile report.

Copyright © 2012, Texas Instruments Incorporated

9.5 Configuration File Specification

The stack depth profiler enables you to specify indirect calls and reentrant procedures using a configuration file. The configuration file is input to the profiler, along with the executable to be analyzed. The profiler applies the configuration information to the stack depth information extracted from the executable and reports the total stack usage.

9.5.1 Specify Indirect Calls

The configuration file is composed of a series of sections delimited by the section type. Each section contains information for a specific section type. As shown in Example 9-3, the section specifying indirectly called functions is delimited by <INDIRECT> and </INDIRECT>, and contains information on the set of procedures called indirectly.

Example 9-3. Specifying Indirect Calls

```
<INDIRECT>
func1: callee1[(mode)],callee2[(mode)],[],calleeN[(mode)]
func2: callee1[(mode)],[],calleeN[(mode)]
[]
funcm: callee1[(mode)],[],calleeN[(mode)]
</INDIRECT>
```

The keywords <INDIRECT> and </INDIRECT> delimit an indirect callee section. Each line in the section begins with the name of a calling procedure, func, followed by a colon (:), which is then followed by a list of functions that are called indirectly from func. Each entry in the indirect callee list consists of the callee name, and an optional mode (in parentheses) in which the callee was compiled.

The profiler recognizes the specification of one of three modes: ARM, THUMB, and DUAL (default). If a mode is not specified, the profiler will assume DUAL mode. For DUAL mode callee functions, the profiler automatically adds the names of the callees (in ARM and THUMB modes) to the callee list for func. In this example, function bar is specified as a callee in DUAL mode, and both _bar and $bar are added to the callee list for main:

```
<INDIRECT>
main:foo(ARM),bar
</INDIRECT>
```

Each indirect call can be specified on a separate line within an indirect call section:

```
<INDIRECT>
main:foo(ARM),
main:bar
</INDIRECT>
```

The profiler performs no semantic checks on the information input via the configuration file. That is, it does not check to determine if the procedures listed in the callee list or the caller list exist or if they exist in the mode specified. The profiler checks, however, for available debug information for a function and generates a warning if none is found.

The profiler declares the following types of loads of PC (in ARM) mode as indirect call points, where #imm is an immediate value.

```
LDR{B,H} PC,[reg,#imm]
```

However, loads of PC with register offsets are not identified as call points since such loads are used generally for switch table jumps.

Copyright © 2012, Texas Instruments Incorporated

9.5.2 Specifying Reentrant Procedures

Reentrant functions present a difficult problem to the profiler's ability to accurately estimate the worst-case space requirement of an application. One difficulty with reentrant functions is that the depth of recursion is often data-dependent. You may not be able to accurately inform the profiler about the depth of recursion without knowing in advance all of the input data sets that will be used in the application. Another difficulty is the presence of multiple recursive functions in a call-graph, which would make it hard to determine the recursive behavior of each function in the call-graph.

In light of these difficulties, no automated analysis is performed to determine the presence of reentrant functions or their nature. Instead, we must input the maximum recursive depth of a function. The profiler assumes that your input is correct and reliable. It issues a warning and warns if there is no apparent recursion in the call graph for a function and recursion depth that has been specified. Beyond the required analysis to issue such a warning, the profiler does not perform any analysis on reentrant calls.

The configuration file issued to specify the list of reentrant procedures in the application. The syntax is:

<REENTRANT>
func1 **(** *depth* **)**
func2 **(** *depth* **)**
</REENTRANT>

The reentrant section is delimited by the <REENTRANT> and </REENTRANT> keywords as shown above. Each line in the section consists of a function name followed by the recursion depth (in parentheses) to be assumed for the function. The depth value should be a valid, non-negative integer.

The profiler multiplies the stack size of the function by its specified depth to obtain the final stack usage requirement for the function. For example,

```
<REENTRANT>
foo(5)
</REENTRANT>
```

For this example, the profiler determines that the static stack usage requirement of foo is 100, the maximum dynamic requirement of foo would be 100*5 = 500. The depth of a function defaults to 1 in the absence of a depth field.

The profiler does not make any effort to determine if the procedure specified in the reentrant section is indeed recursive. It assumes that the input is correct and reliable.

9.5.3 Specifying Interrupt Service Routines

Interrupt service routines (ISRs), are used to handle interrupts. An interrupt service routine is a special root function that is usually not directly called from the application. In ARM, an interrupt request can be serviced by an ISR in any of the six modes. Each mode typically uses a different stack area for the interrupt handler. The SDA can output the stack requirements of an ISR if you identify ISRs and their modes of operations. This is the syntax for specifying an ISR and its mode:

<INTERRUPT>
routine_name **(** *mode* **)**
</INTERRUPT>

In the syntax, *routine_name* is the name of the ISR and *mode* can be one of the following:

irq	general purpose interrupt
fiq	data transfer or channel process
svc	operating system protected mode
abt	data or instruction prefetch abort
sys	privileged user mode for the operating system
und	undefined instruction

Copyright © 2012, Texas Instruments Incorporated

The stack space requirement for ISRs is output after the stack space requirements for routines directly called by the application.

9.5.4 Other Features

The configuration file supports all preprocessing directives supported by armcl. The configuration file should be processed using the armcl preprocessor before being input to the profiler. For example, file config1 contains the following <INDIRECT> lines and file config2 contains the following <REENTRANT> lines:

```
<INDIRECT>
main:foo,
</INDIRECT>
#include "config2"
```

```
<REENTRANT>
foo(5)
</REENTRANT>
```

The armcl preprocessor can be used to combine the two config files. For example:

```
armcl --preproc_only config1
```

The preprocessor adds or replaces the file suffix with .pp. In the above example, the combined config files will be found in the file "config1.pp". The profiler supports C++ style comments. All characters following the // symbol to the end of the line (new line character) are ignored. The // can appear anywhere in a configuration file. For example:

```
<INDIRECT>
// This is a comment
main:foo
// End of indirect callee list
</INDIRECT>
```

The configuration file does not permit nesting of sections. For example, the following specification is illegal:

```
<INDIRECT>
foo:bar
<REENTRANT>
foo(5)
</REENTRANT>
</INDIRECT>
```

The profiler performs no semantic checks on the information specified in the configuration file and assumes that the information provided is correct and valid. It is your responsibility to ensure the validity of information specified in the configuration file.

9.5.5 Tail Calls

A tail call is code that branches to a routine foo() at the end of routine bar() so that control returns directly to bar's caller at the end of foo()). The SDA handles tail calls by identifying tail call points as both call point and return points. The pseudo assembly instruction CRET is used for representing tail calls.

Copyright © 2012, Texas Instruments Incorporated

9.6 Profiler Generated Warnings

The static stack depth profiler detects the following situations and issues a warning in each case:

- A function contains an indirect branch (branch to the contents of a register) and there are no <INDIRECT>entries for the function in the configuration file, the profiler issues a warning that a potential indirect call has been observed and this caller/callee pair is not specified in the configuration file. Not all indirect branches are function calls; e.g., in 32-BIS mode, long branches are achieved through indirect branches.
- A set of indirect callees for a function is not specified and the profiler observes no indirect branches in the function, it issues a warning indicating the absence of indirect calls.
- A function's call graph has a link back to the function indicating potential recursion and no recursion depth is specified, the profiler issues a warning indicating that the function is reentrant. A warning and not an error is issued since a function could have an apparent recursion and not a real one.
- The profiler discovers no links back to the function in a call-graph and a recursion depth is specified, it issues a warning.
- No stack usage information for a function is found.
- No debug information for a function is found.
- The profiler generates a potential indirect callee warning for each routine compiled in dual mode. This is because, in dual mode, each call to an external routine is performed by jumping to the RTS routine IND_CALL or IND$CALL (depending on the mode of the caller).
- The profiler generates potential indirect callee warnings for routines in the RTS library.

9.7 Run-Time Support for Stack Depth Profiling

The ARM compiler provides a debug option to determine the maximum stack usage of an application. If the application is compiled with the -debug:sdp option, the compiler places a call to a stack depth bookkeeping routine (C_SDPBK for 32-BIS and C$SDPBK for 16-BIS), immediately after the frame has been allocated for a function. This bookkeeping routine tracks the maximum stack usage for each function. There is a profiled version of the run-time support library that can be linked when -debug:sdp is turned on. This enables you to determine the complete stack usage of an application that contains run-time support utilities (such as I/O utilities).

There is an overhead associated with the -debug:sdp option. The link register (LR) is always saved when this option is turned on, since all functions in the application call the bookkeeping routine.

Copyright © 2012, Texas Instruments Incorporated

TEXAS INSTRUMENTS

Glossary

absolute lister— A debugging tool that allows you to create assembler listings that contain absolute addresses.

assignment statement— A statement that initializes a variable with a value.

autoinitialization— The process of initializing global C variables (contained in the .cinit section) before program execution begins.

autoinitialization at run time— An autoinitialization method used by the linker when linking C code. The linker uses this method when you invoke it with the --rom_model link option. The linker loads the .cinit section of data tables into memory, and variables are initialized at run time.

alias disambiguation— A technique that determines when two pointer expressions cannot point to the same location, allowing the compiler to freely optimize such expressions.

aliasing— The ability for a single object to be accessed in more than one way, such as when two pointers point to a single object. It can disrupt optimization, because any indirect reference could refer to any other object.

allocation— A process in which the linker calculates the final memory addresses of output sections.

ANSI— American National Standards Institute; an organization that establishes standards voluntarily followed by industries.

archive library— A collection of individual files grouped into a single file by the archiver.

archiver— A software program that collects several individual files into a single file called an archive library. With the archiver, you can add, delete, extract, or replace members of the archive library.

assembler— A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro definitions. The assembler substitutes absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

assignment statement— A statement that initializes a variable with a value.

autoinitialization— The process of initializing global C variables (contained in the .cinit section) before program execution begins.

autoinitialization at run time— An autoinitialization method used by the linker when linking C code. The linker uses this method when you invoke it with the --rom_model link option. The linker loads the .cinit section of data tables into memory, and variables are initialized at run time.

big endian— An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

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.

Copyright © 2012, Texas Instruments Incorporated

byte— Per ANSI/ISO C, the smallest addressable unit that can hold a character.

C/C++ compiler— A software program that translates C source statements into assembly language source statements.

code generator— A compiler tool that takes the file produced by the parser or the optimizer and produces an assembly language source file.

COFF— Common object file format; a system of object files configured according to a standard developed by AT&T. These files are relocatable in memory space.

command file— A file that contains options, filenames, directives, or commands for the linker or hex conversion utility.

comment— A source statement (or portion of a source statement) that documents or improves readability of a source file. Comments are not compiled, assembled, or linked; they have no effect on the object file.

compiler program— A utility that lets you compile, assemble, and optionally link in one step. The compiler runs one or more source modules through the compiler (including the parser, optimizer, and code generator), the assembler, and the linker.

configured memory— Memory that the linker has specified for allocation.

constant— A type whose value cannot change.

cross-reference listing— An output file created by the assembler that lists the symbols that were defined, what line they were defined on, which lines referenced them, and their final values.

.data section— One of the default object file sections. The .data section is an initialized section that contains initialized data. You can use the .data directive to assemble code into the .data section.

direct call— A function call where one function calls another using the function's name.

directives— Special-purpose commands that control the actions and functions of a software tool (as opposed to assembly language instructions, which control the actions of a device).

disambiguation— See *alias disambiguation*

dynamic memory allocation— A technique used by several functions (such as malloc, calloc, and realloc) to dynamically allocate memory for variables at run time. This is accomplished by defining a large memory pool (heap) and using the functions to allocate memory from the heap.

ELF— Executable and Linkable Format; a system of object files configured according to the System V Application Binary Interface specification.

emulator— A hardware development system that duplicates the 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.

Copyright © 2012, Texas Instruments Incorporated

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, that lists source statements, their line numbers, and their effects on the section program counter (SPC).

little endian— An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

loader— A device that places an executable object file into system memory.

loop unrolling— An optimization that expands small loops so that each iteration of the loop appears in your code. Although loop unrolling increases code size, it can improve the performance of your code.

macro— A user-defined routine that can be used as an instruction.

macro call— The process of invoking a macro.

Copyright © 2012, Texas Instruments Incorporated

macro definition— A block of source statements that define the name and the code that make up a macro.

macro expansion— The process of inserting source statements into your code in place of a macro call.

map file— An output file, created by the linker, that shows the memory configuration, section composition, section allocation, symbol definitions and the addresses at which the symbols were defined for your program.

memory map— A map of target system memory space that is partitioned into functional blocks.

name mangling— A compiler-specific feature that encodes a function name with information regarding the function's arguments return types.

object file— An assembled or linked file that contains machine-language object code.

object library— An archive library made up of individual object files.

operand— An argument of an assembly language instruction, assembler directive, or macro directive that supplies information to the operation performed by the instruction or directive.

optimizer— A software tool that improves the execution speed and reduces the size of C programs.

options— Command-line parameters that allow you to request additional or specific functions when you invoke a software tool.

output section— A final, allocated section in a linked, executable module.

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, that contains the source for the run time-support functions.

Copyright © 2012, Texas Instruments Incorporated

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 a simulator or an emulator.

target system— The system on which the object code you have developed is executed.

.text section— One of the default object file sections. The .text section is initialized and contains executable code. You can use the .text directive to assemble code into the .text section.

trigraph sequence— A 3-character sequence that has a meaning (as defined by the ISO 646-1983 Invariant Code Set). These characters cannot be represented in the C character set and are expanded to one character. For example, the trigraph ??' is expanded to ^.

trip count— The number of times that a loop executes before it terminates.

unconfigured memory— Memory that is not defined as part of the memory map and cannot be loaded with code or data.

uninitialized section— A object file section that reserves space in the memory map but that has no actual contents. These sections are built with the .bss and .usect directives.

unsigned value— A value that is treated as a nonnegative number, regardless of its actual sign.

variable— A symbol representing a quantity that can assume any of a set of values.

veneer— A sequence of instructions that serves as an alternate entry point into a routine if a state change is required.

word— A 32-bit addressable location in target memory

Copyright © 2012, Texas Instruments Incorporated

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components which meet ISO/TS16949 requirements, mainly for automotive use. Components which have not been so designated are neither designed nor intended for automotive use; and TI will not be responsible for any failure of such components to meet such requirements.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

Copyright © 2012, Texas Instruments Incorporated