C7000 Optimizing C/C++ Compiler v6.2.x.LTS

User's Guide

Literature Number: SPRUIG8L
JANUARY 2018 – REVISED NOVEMBER 2026

Table of Contents

Read This First 13
About This Manual 13
Notational Conventions 13
Related Documentation 14
Related Documentation From Texas Instruments 15
Trademarks 15
1 Introduction to the Software Development Tools 16
1.1 Software Development Tools Overview 17
1.2 Compiler Interface 18
1.3 ANSI/ISO Standard 19
1.4 Output Files 19
2 Getting Started with the Code Generation Tools 20
2.1 How Code Composer Studio Projects Use the Compiler 21
2.2 Compiling from the Command Line 21
3 Using the C/C++ Compiler 22
3.1 About the Compiler 23
3.2 Invoking the C/C++ Compiler 23
3.3 Changing the Compiler's Behavior with Options 24
3.3.1 Linker Options 30
3.3.2 Frequently Used Options 32
3.3.3 Miscellaneous Useful Options 33
3.3.4 Run-Time Model Options 34
3.3.5 Selecting Target CPU Version (--silicon_version Option) 35
3.3.6 Symbolic Debugging and Profiling Options 35
3.3.7 Specifying Filenames 35
3.3.8 Changing How the Compiler Interprets Filenames 36
3.3.9 Changing How the Compiler Processes C Files 36
3.3.10 Changing How the Compiler Interprets and Names Extensions 36
3.3.11 Specifying Directories 37
3.4 Controlling the Compiler Through Environment Variables 37
3.4.1 Setting Default Compiler Options (C7X_C_OPTION) 37
3.4.2 Naming One or More Alternate Directories (C7X_C_DIR) 38
3.4.3 Controlling the Location of Temporary Files 39
3.5 Controlling the Preprocessor 39
3.5.1 Predefined Macro Names 39
3.5.2 The Search Path for #include Files 41
3.5.3 Support for the #warning and #warn Directives 42
3.5.4 Generating a Preprocessed Listing File (--preproc_only Option) 42
3.5.5 Continuing Compilation After Preprocessing (--preproc_with_compile Option) 42
3.5.6 Generating a Preprocessed Listing File with Comments (--preproc_with_comment Option) 43
3.5.7 Generating Preprocessed Listing with Line-Control Details (--preproc_with_line Option) 43
3.5.8 Generating Preprocessed Output for a Make Utility (--preproc_dependency Option) 43
3.5.9 Generating a List of Files Included with #include (--preproc_includes Option) 43
3.5.10 Generating a List of Macros in a File (--preproc_macros Option) 43
3.6 Passing Arguments to main() 44
3.7 Understanding Diagnostic Messages 44
3.7.1 Controlling Diagnostic Messages 46
3.7.2 How You Can Use Diagnostic Suppression Options 47
3.8 Other Messages 47

Copyright © 2026 Texas Instruments Incorporated

3.9 Generating a Raw Listing File (--gen_preprocessor_listing Option) ... 47
3.10 Using Inline Function Expansion ... 48
3.10.1 Inlining Intrinsic Operators ... 49
3.10.2 Inlining Restrictions ... 50
3.10.3 Unguarded Definition-Controlled Inlining ... 50
3.10.4 Guarded Inlining and the _INLINE Preprocessor Symbol ... 51
3.11 Using Interlist ... 52
3.12 About the Application Binary Interface ... 53
3.13 Enabling Entry Hook and Exit Hook Functions ... 54
4 Optimizing Your Code ... 55
4.1 Invoking Optimization ... 56
4.2 Controlling Code Size Versus Speed ... 57
4.3 Performing File-Level Optimization (--opt_level=3 option) ... 57
4.3.1 Creating an Optimization Information File (--gen_opt_info Option) ... 57
4.4 Program-Level Optimization (--program_level_compile and --opt_level=3 options) ... 58
4.4.1 Controlling Program-Level Optimization (--call_assumptions Option) ... 59
4.5 Automatic Inline Expansion (--auto_inline Option) ... 60
4.6 Link-Time Optimization (--opt_level=4 Option) ... 61
4.6.1 Option Handling ... 61
4.6.2 Incompatible Types ... 61
4.7 Optimizing Software Pipelining ... 62
4.7.1 Turn Off Software Pipelining (--disable_software_pipeline Option) ... 62
4.7.2 Software Pipelining Information ... 62
4.7.3 Collapsing Prologs and Epilogs for Improved Performance and Code Size ... 66
4.8 Redundant Loops ... 67
4.9 Indicating Whether Certain Aliasing Techniques Are Used ... 68
4.9.1 Use the --aliased_variables Option When Certain Aliases are Used ... 68
4.10 Prevent Reordering of Associative Floating-Point Operations ... 68
4.11 Using Performance Advice to Optimize Code ... 69
4.11.1 Advice #35000: Use restrict to improve loop performance ... 69
4.12 Using the Interlist Feature With Optimization ... 70
4.13 Debugging and Profiling Optimized Code ... 70
4.13.1 Profiling Optimized Code ... 70
4.14 What Kind of Optimization Is Being Performed? ... 71
4.14.1 Cost-Based Register Allocation ... 71
4.14.2 Alias Disambiguation ... 71
4.14.3 Branch Optimizations and Control-Flow Simplification ... 72
4.14.4 Data Flow Optimizations ... 72
4.14.5 Expression Simplification ... 72
4.14.6 Inline Expansion of Functions ... 72
4.14.7 Function Symbol Aliasing ... 73
4.14.8 Induction Variables and Strength Reduction ... 73
4.14.9 Loop-Invariant Code Motion ... 73
4.14.10 Loop Rotation ... 73
4.14.11 Loop Collapsing and Loop Coalescing ... 73
4.14.12 Unroll-and-jam ... 74
4.14.13 Vectorization (SIMD) ... 74
4.14.14 Instruction Scheduling ... 74
4.14.15 Register Variables ... 74
4.14.16 Register Tracking/Targeting ... 74
4.14.17 Software Pipelining ... 74
4.15 Streaming Engine and Streaming Address Generator ... 75
4.15.1 Streaming Engine Overview ... 75
4.15.2 Theory of Operation for Streaming Engines and Streaming Address Generators ... 75
4.15.3 Streaming Address Generator Overview ... 76
4.15.4 Advantages of Using Streaming Engines and Streaming Address Generators ... 76
4.15.5 Interface for Streaming Engine and Streaming Address Generator ... 77
4.15.6 Parameter Template Configuration ... 77
4.15.7 Using the Streaming Engine ... 79
4.15.8 Using the Streaming Address Generator ... 80
4.15.9 Automatic Use of Streaming Engine and Streaming Address Generator (--auto_stream Option) ... 84

Copyright © 2026 Texas Instruments Incorporated

4.16 Nested Loop Controller (NLC)....86
4.16.1 Obstacles That May Inhibit Use of NLC....86
5 C/C++ Language Implementation....87
5.1 Characteristics of C7000 C....88
5.1.1 Implementation-Defined Behavior....88
5.2 Characteristics of C7000 C++....92
5.3 Data Types....93
5.3.1 Size of Enum Types....94
5.3.2 Vector, Complex, and Complex Vector Data Types....95
5.4 File Encodings and Character Sets....98
5.5 Keywords....98
5.5.1 The complex Keyword....99
5.5.2 The const Keyword....99
5.5.3 The __cregister Keyword....100
5.5.4 The restrict Keyword....103
5.5.5 The volatile Keyword....103
5.6 C++ Exception Handling....104
5.7 Register Variables and Parameters....105
5.8 Pragma Directives....106
5.8.1 The CALLS Pragma....107
5.8.2 The CLINK Pragma....107
5.8.3 The COALESCE_LOOP Pragma....107
5.8.4 The CODE_ALIGN Pragma....108
5.8.5 The CODE_SECTION Pragma....108
5.8.6 The DATA_ALIGN Pragma....109
5.8.7 The DATA_MEM_BANK Pragma....109
5.8.8 The DATA_SECTION Pragma....110
5.8.9 The Diagnostic Message Pragmas....111
5.8.10 The FORCEINLINE Pragma....112
5.8.11 The FORCEINLINE_RECURSIVE Pragma....112
5.8.12 The FUNC_ALWAYS_INLINE Pragma....113
5.8.13 The FUNC_CANNOT_INLINE Pragma....113
5.8.14 The FUNC_EXT_CALLED Pragma....114
5.8.15 The FUNC_IS_PURE Pragma....114
5.8.16 The FUNC_IS_SYSTEM Pragma....114
5.8.17 The FUNC_NEVER_RETURNS Pragma....115
5.8.18 The FUNC_NO_GLOBAL_ASG Pragma....115
5.8.19 The FUNC_NO_IND_ASG Pragma....115
5.8.20 The FUNCTION_OPTIONS Pragma....115
5.8.21 The INTERRUPT Pragma....116
5.8.22 The LOCATION Pragma....117
5.8.23 The MUST_ITERATE Pragma....117
5.8.24 The NOINIT and PERSISTENT Pragmas....119
5.8.25 The NOINLINE Pragma....120
5.8.26 The NO_COALESCE_LOOP Pragma....120
5.8.27 The NO_HOOKS Pragma....120
5.8.28 The once Pragma....121
5.8.29 The pack Pragma....121
5.8.30 The PROB_ITERATE Pragma....122
5.8.31 The RETAIN Pragma....122
5.8.32 The SET_CODE_SECTION and SET_DATA_SECTION Pragmas....123
5.8.33 The STRUCT_ALIGN Pragma....124
5.8.34 The UNROLL Pragma....124
5.8.35 The WEAK Pragma....125
5.9 The _Pragma Operator....126
5.10 Application Binary Interface....127
5.11 Object File Symbol Naming Conventions (Linknames)....127
5.12 Changing the ANSI/ISO C/C++ Language Mode....128
5.12.1 C99 Support (--c99)....128
5.12.2 C11 Support (--c11)....128
5.12.3 Strict ANSI Mode and Relaxed ANSI Mode (--strict_ansi and --relaxed_ansi)....129

Copyright © 2026 Texas Instruments Incorporated

5.13 GNU and Clang Language Extensions ... 130
5.13.1 Extensions ... 130
5.13.2 Function Attributes ... 132
5.13.3 For Loop Attributes ... 133
5.13.4 Variable Attributes ... 133
5.13.5 Type Attributes ... 134
5.13.6 Built-In Functions ... 135
5.14 Operations and Functions for Vector Data Types ... 136
5.14.1 Vector Literals and Concatenation ... 136
5.14.2 Unary and Binary Operators for Vectors ... 137
5.14.3 Ternary Operators for Vectors (?:) ... 142
5.14.4 Swizzle Operators for Vectors ... 142
5.14.5 Unsupported Vector Comparison Operators ... 144
5.14.6 Conversion Functions for Vectors ... 145
5.14.7 Re-Interpretation Functions for Vectors ... 145
5.14.8 Vector Predicate Type ... 145
5.15 C7000 Intrinsics ... 148
5.15.1 High-Level Overloaded Intrinsics ... 148
5.15.2 Intrinsics Defined for Special Load and Store Instructions ... 149
5.15.3 Direct-Mapped Intrinsics ... 150
5.15.4 Lookup Table and Histogram Intrinsics ... 150
5.15.5 Matrix-Multiply Accelerator (MMA) Intrinsics ... 150
5.15.6 Legacy Intrinsics ... 150
5.16 C7000 Scalable Vector Programming ... 151
6 Run-Time Environment ... 154
6.1 Memory ... 155
6.1.1 Sections ... 155
6.1.2 C/C++ System Stack ... 156
6.1.3 Dynamic Memory Allocation ... 157
6.2 Object Representation ... 158
6.2.1 Data Type Storage ... 158
6.2.2 Bit Fields ... 163
6.2.3 Character String Constants ... 164
6.3 Register Conventions ... 165
6.4 Function Structure and Calling Conventions ... 167
6.4.1 How a Function Makes a Call ... 167
6.4.2 How a Called Function Responds ... 168
6.4.3 Accessing Arguments and Local Variables ... 169
6.5 Accessing Linker Symbols in C and C++ ... 169
6.6 Run-Time-Support Arithmetic Routines ... 169
6.7 System Initialization ... 171
6.7.1 Boot Hook Functions for System Pre-Initialization ... 171
6.7.2 Automatic Initialization of Variables ... 171
7 Using Run-Time-Support Functions and Building Libraries ... 176
7.1 C and C++ Run-Time Support Libraries ... 177
7.1.1 Linking Code With the Object Library ... 177
7.1.2 Header Files ... 177
7.1.3 Modifying a Library Function ... 178
7.1.4 Support for String Handling ... 179
7.1.5 Minimal Support for Internationalization ... 179
7.1.6 Support for Time and Clock Functions ... 179
7.1.7 Allowable Number of Open Files ... 180
7.1.8 Library Naming Conventions ... 180
7.2 The C I/O Functions ... 181
7.2.1 High-Level I/O Functions ... 181
7.2.2 Overview of Low-Level I/O Implementation ... 182
7.2.3 Device-Driver Level I/O Functions ... 187
7.2.4 Adding a User-Defined Device Driver for C I/O ... 192
7.2.5 The device Prefix ... 193
7.3 Handling Reentrancy (_register_lock() and _register_unlock() Functions) ... 195
7.4 Library-Build Process ... 196

Copyright © 2026 Texas Instruments Incorporated

7.4.1 Required Non-Texas Instruments Software....196
7.4.2 Using the Library-Build Process....196
7.4.3 Extending mklib....199
8 Introduction to Object Modules....200
8.1 Object File Format Specifications....201
8.2 Executable Object Files....201
8.3 Introduction to Sections....201
8.3.1 Special Section Names....202
8.4 How the Linker Handles Sections....202
8.4.1 Combining Input Sections....203
8.4.2 Placing Sections....203
8.5 Symbols....203
8.5.1 Local Symbols....204
8.5.2 Weak Symbols....204
8.6 Loading a Program....204
9 Program Loading and Running....205
9.1 Loading....206
9.2 Entry Point....206
9.3 Run-Time Initialization....207
9.3.1 The _c_int00 Function....207
9.3.2 RAM Model vs. ROM Model....207
9.3.3 About Linker-Generated Copy Tables....209
9.4 Arguments to main....210
9.5 Run-Time Relocation....210
9.6 Additional Information....210
10 Archiver Description....211
10.1 Archiver Overview....212
10.2 The Archiver's Role in the Software Development Flow....212
10.3 Invoking the Archiver....213
10.4 Archiver Examples....214
10.5 Library Information Archiver Description....215
10.5.1 Invoking the Library Information Archiver....215
10.5.2 Library Information Archiver Example....216
10.5.3 Listing the Contents of an Index Library....216
10.5.4 Requirements....216
11 Linking C/C++ Code....217
11.1 Invoking the Linker Through the Compiler (-z Option)....218
11.1.1 Invoking the Linker Separately....218
11.1.2 Invoking the Linker as Part of the Compile Step....219
11.1.3 Disabling the Linker (--compile_only Compiler Option)....219
11.2 Linker Code Optimizations....220
11.2.1 Conditional Linking....220
11.2.2 Generating Function Subsections (--gen_func_subsections Compiler Option)....220
11.2.3 Generating Aggregate Data Subsections (--gen_data_subsections Compiler Option)....220
11.3 Controlling the Linking Process....221
11.3.1 Including the Run-Time-Support Library....221
11.3.2 Run-Time Initialization....222
11.3.3 Global Object Constructors....222
11.3.4 Specifying the Type of Global Variable Initialization....223
11.3.5 Specifying Where to Allocate Sections in Memory....223
11.3.6 A Sample Linker Command File....224
12 Linker Description....225
12.1 Linker Overview....226
12.2 The Linker's Role in the Software Development Flow....226
12.3 Invoking the Linker....227
12.4 Linker Options....228
12.4.1 Wildcards in File, Section, and Symbol Patterns....230
12.4.2 Specifying C/C++ Symbols with Linker Options....230
12.4.3 Relocation Capabilities (--absolute_exe and --relocatable Options)....230
12.4.4 Allocate Memory for Use by the Loader to Pass Arguments (--arg_size Option)....231
12.4.5 Compression (--cinit_compression and --copy_compression Option)....231

Copyright © 2026 Texas Instruments Incorporated

12.4.6 Compress DWARF Information (--compress_dwarf Option) 232
12.4.7 Control Linker Diagnostics 232
12.4.8 Automatic Library Selection (--disable_auto_rts Option) 232
12.4.9 Do Not Remove Unused Sections (--unused_section_elimination Option) 232
12.4.10 Linker Command File Preprocessing (--disable_pp, --define and --undefine Options) 233
12.4.11 Define an Entry Point (--entry_point Option) 234
12.4.12 Set Default Fill Value (--fill_value Option) 234
12.4.13 Define Heap Size (--heap_size Option) 235
12.4.14 Hiding Symbols 235
12.4.15 Alter the Library Search Algorithm (--library, --search_path, and C7X_C_DIR) 235
12.4.16 Change Symbol Localization 239
12.4.17 Create a Map File (--map_file Option) 240
12.4.18 Manage Map File Contents (--mapfile_contents Option) 241
12.4.19 Disable Name Demangling (--no_demangle) 242
12.4.20 Merging of Symbolic Debugging Information 242
12.4.21 Strip Symbolic Information (--no_symtable Option) 242
12.4.22 Name an Output Module (--output_file Option) 243
12.4.23 Prioritizing Function Placement (--preferred_order Option) 243
12.4.24 C Language Options (--ram_model and --rom_model Options) 243
12.4.25 Retain Discarded Sections (--retain Option) 243
12.4.26 Scan All Libraries for Duplicate Symbol Definitions (--scan_libraries) 244
12.4.27 Define Stack Size (--stack_size Option) 244
12.4.28 Mapping of Symbols (--symbol_map Option) 244
12.4.29 Generate Far Call Trampolines (--trampolines Option) 245
12.4.30 Introduce an Unresolved Symbol (--undef_sym Option) 247
12.4.31 Display a Message When an Undefined Output Section Is Created (--warn_sections) 247
12.4.32 Generate XML Link Information File (--xml_link_info Option) 247
12.4.33 Zero Initialization (--zero_init Option) 248
12.5 Linker Command Files 249
12.5.1 Reserved Names in Linker Command Files 250
12.5.2 Constants in Linker Command Files 250
12.5.3 Accessing Files and Libraries from a Linker Command File 250
12.5.4 The MEMORY Directive 252
12.5.5 The SECTIONS Directive 254
12.5.6 Placing a Section at Different Load and Run Addresses 266
12.5.7 Using GROUP and UNION Statements 267
12.5.8 Special Section Types (DSECT, COPY, NOLOAD, and NOINIT) 271
12.5.9 Assigning Symbols at Link Time 271
12.5.10 Creating and Filling Holes 277
12.6 Linker Symbols 280
12.6.1 Linker-Defined Functions and Arrays 280
12.6.2 Linker-Defined Integer Values 280
12.6.3 Linker-Defined Addresses 280
12.6.4 More About the _symval Operator 281
12.6.5 Using _symval, PC-Relative Addressing, and Far Data 281
12.6.6 Weak Symbols 282
12.6.7 Resolving Symbols with Object Libraries 282
12.7 Default Placement Algorithm 284
12.7.1 How the Allocation Algorithm Creates Output Sections 284
12.7.2 Reducing Memory Fragmentation 284
12.8 Using Linker-Generated Copy Tables 285
12.8.1 Using Copy Tables for Boot Loading 285
12.8.2 Using Built-in Link Operators in Copy Tables 285
12.8.3 Overlay Management Example 286
12.8.4 Generating Copy Tables With the table() Operator 287
12.8.5 Compression 290
12.8.6 Copy Table Contents 294
12.8.7 General Purpose Copy Routine 295
12.9 Partial (Incremental) Linking 296
12.10 Linking C/C++ Code 296
12.10.1 Run-Time Initialization 297

Copyright © 2026 Texas Instruments Incorporated

12.10.2 Object Libraries and Run-Time Support ... 297
12.10.3 Setting the Size of the Stack and Heap Sections ... 297
12.10.4 Initializing and Autoinitializing Variables at Run Time ... 297
12.10.5 Constraints Due to CMMU Configuration ... 297
12.11 Linker Example ... 298
12.12 Static Shared Objects in Multicore Applications ... 301
12.12.1 SSO Example ... 302
12.12.2 Step 1: Compile and Link Applications ... 302
12.12.3 Step 2: Identify Common Objects ... 303
12.12.4 Step 3: Create a Static Shared Object (SSO) File ... 305
12.12.5 Step 4: Relink Applications Against the SSO ... 307
12.12.6 Summary of SSO Command Line Options ... 309
13 Object File Utilities ... 310
13.1 Invoking the Object File Display Utility ... 311
13.2 Invoking the Disassembler ... 312
13.3 Invoking the Name Utility ... 313
13.4 Invoking the Strip Utility ... 313
14 C++ Name Demangler ... 314
14.1 Invoking the C++ Name Demangler ... 315
14.2 Sample Usage of the C++ Name Demangler ... 316
A XML Link Information File Description ... 317
A.1 XML Information File Element Types ... 318
A.2 Document Elements ... 318
A.2.1 Header Elements ... 318
A.2.2 Input File List ... 319
A.2.3 Object Component List ... 320
A.2.4 Logical Group List ... 321
A.2.5 Placement Map ... 323
A.2.6 Far Call Trampoline List ... 324
A.2.7 Symbol Table ... 325
B Unsupported Tools and Features ... 326
B.1 List of Unsupported Tools and Features ... 326
C Glossary ... 327
D Revision History ... 334

List of Figures

Figure 1-1. C7000 Software Development Flow ... 17
Figure 4-1. Software-Pipelined Loop ... 62
Figure 5-1. Memory and Register Layout of Complex Components ... 141
Figure 6-1. Char and Short Data Storage Format ... 159
Figure 6-2. 32-Bit Data Storage Format ... 160
Figure 6-3. 64-Bit Data Storage Format Signed 64-bit long ... 161
Figure 6-4. Unsigned 64-bit long ... 161
Figure 6-5. Single-Precision Floating-Point Char Data Storage Format ... 162
Figure 6-6. Bit-Field Packing in Big-Endian and Little-Endian Formats ... 164
Figure 6-7. Autoinitialization at Run Time ... 172
Figure 6-8. Initialization at Load Time ... 175
Figure 6-9. Constructor Table ... 175
Figure 8-1. Partitioning Memory Into Logical Blocks ... 202
Figure 8-2. Combining Input Sections to Form an Executable Object Module ... 203
Figure 9-1. Autoinitialization at Run Time ... 208
Figure 9-2. Initialization at Load Time ... 208
Figure 10-1. The Archiver in the C7000 Software Development Flow ... 212
Figure 12-1. The Linker in the C7000 Software Development Flow ... 226
Figure 12-2. Section Placement Defined by the SECTIONS Directive Example ... 256
Figure 12-3. Memory Allocation Shown in The UNION Statement and Separate Load Addresses for UNION Sections ... 268
Figure 12-4. Compressed Copy Table ... 291
Figure 12-5. Handler Table ... 292

Copyright © 2026 Texas Instruments Incorporated

List of Tables

Table 2-1. Steps for Creating a CCS Project ... 21
Table 3-1. Processor Options ... 24
Table 3-2. Optimization Options(1) ... 24
Table 3-3. Advanced Optimization Options(1) ... 25
Table 3-4. Debug Options ... 26
Table 3-5. Include Options ... 26
Table 3-6. Control Options ... 26
Table 3-7. Language Options ... 26
Table 3-8. Parser Preprocessing Options ... 27
Table 3-9. Predefined Macro Options ... 27
Table 3-10. Diagnostic Message Options ... 27
Table 3-11. Supplemental Information Options ... 28
Table 3-12. Run-Time Model Options ... 28
Table 3-13. Entry/Exit Hook Options ... 28
Table 3-14. Assembly Options ... 28
Table 3-15. File Type Specifier Options ... 28
Table 3-16. Directory Specifier Options ... 29
Table 3-17. Default File Extensions Options ... 29
Table 3-18. Command Files Options ... 29
Table 3-19. Performance Advisor Options ... 29
Table 3-20. Linker Basic Options ... 30
Table 3-21. File Search Path Options ... 30
Table 3-22. Command File Preprocessing Options ... 30
Table 3-23. Diagnostic Message Options ... 30
Table 3-24. Linker Output Options ... 31
Table 3-25. Symbol Management Options ... 31
Table 3-26. Run-Time Environment Options ... 31
Table 3-27. Miscellaneous Options ... 31
Table 3-28. Predefined C7000 Macro Names ... 39
Table 3-29. Raw Listing File Identifiers ... 48
Table 3-30. Raw Listing File Diagnostic Identifiers ... 48
Table 4-1. Optimization Levels ... 57
Table 4-2. Options That You Can Use With --opt_level=3 ... 57
Table 4-3. Selecting a Level for the --gen_opt_info Option ... 57
Table 4-4. Selecting a Level for the --call_assumptions Option ... 59
Table 4-5. Special Considerations When Using the --call_assumptions Option ... 59
Table 5-1. C7000 C/C++ Data Types ... 93
Table 5-2. Minimum Positive Floating-Point Values ... 94
Table 5-3. C Language Standard Types ... 94
Table 5-4. Vector Data Types ... 96
Table 5-5. Complex Data Types ... 97
Table 5-6. Complex Vector Data Types ... 97
Table 5-7. Vector Widths ... 97
Table 5-8. Control Registers for C7000 ... 100
Table 5-9. GCC Language Extensions ... 130
Table 5-10. Unary and Postfix Operators Supported for Vector Types ... 137
Table 5-11. Binary Operators Supported for Vector Types ... 137
Table 5-12. Legacy Logical and Boolean Vector Operations ... 147
Table 5-13. Macros for Use with MMA and Scalable Vector Programming ... 152
Table 6-1. Data Representation in Registers and Memory ... 158
Table 6-2. Register Usage ... 165
Table 6-3. C7000 Run-Time-Support Arithmetic Functions ... 170
Table 7-1. The mklib Program Options ... 198
Table 11-1. Initialized Sections Created by the Compiler ... 223
Table 11-2. Uninitialized Sections Created by the Compiler ... 223
Table 12-1. Basic Options Summary ... 228
Table 12-2. File Search Path Options Summary ... 228
Table 12-3. Command File Preprocessing Options Summary ... 228

Copyright © 2026 Texas Instruments Incorporated

Table 12-4. Diagnostic Options Summary 228
Table 12-5. Linker Output Options Summary 229
Table 12-6. Symbol Management Options Summary 229
Table 12-7. Run-Time Environment Options Summary 229
Table 12-8. Link-Time Optimization Options Summary 229
Table 12-9. Miscellaneous Options Summary 230
Table 12-10. Predefined C7000 Macro Names 233
Table 12-11. Groups of Operators Used in Expressions (Precedence) 272

Copyright © 2026 Texas Instruments Incorporated

This page intentionally left blank.

Copyright © 2026 Texas Instruments Incorporated

Read This First

About This Manual

The *C7000 Optimizing C/C++ Compiler User's Guide* explains how to use the following Texas Instruments Code Generation compiler tools:

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

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

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

This document describes support for the C7100 variant of the C7000™ processor series.

Notational Conventions

This document uses the following conventions:

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

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

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

cl7x [*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:

cl7x --run_linker {**--rom_model** | **--ram_model**} *filenames* [**--output_file=** *name.out*]
--library= *libraryname*

Copyright © 2026 Texas Instruments Incorporated

Related Documentation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Copyright © 2026 Texas Instruments Incorporated

Related Documentation From Texas Instruments

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

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

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

SPRUIG5 ***C6000-to-C7000 Migration Users Guide***

SPRUIV4 ***C7000 C/C++ Optimization Guide***

SPRUIG3 ***C7000 VCOP Kernel-C Translation Functional Specification***

SPRUIG6 ***C7000 Host Emulation Users Guide***

SPRUIU4 ***C7x Instruction Guide***

(available through your TI Field Application Engineer)

SPRUIP0 ***C71x DSP CPU, Instruction Set, and Matrix Multiply Accelerator Technical Reference Manual***

(available through your TI Field Application Engineer)

SPRUIQ3 ***C71x DSP Corepac Technical Reference Manual***

(available through your TI Field Application Engineer)

SPRAAB5 ***The Impact of DWARF on TI Object Files.***

Trademarks

C7000™, TMS320C6000™, and Code Composer Studio™ are trademarks of Texas Instruments.
OpenCL™ is a trademark of Apple Inc. used by permission by Khronos.
All trademarks are the property of their respective owners.

Copyright © 2026 Texas Instruments Incorporated

Chapter 1

Introduction to the Software Development Tools

The C7000™ is supported by a set of software development tools, which includes an optimizing C/C++ compiler, a linker, and assorted utilities.

This chapter provides an overview of these tools and introduces the features of the optimizing C/C++ compiler.

The C7000™ Code Generation Tools are similar to those provided for the TMS320C6000™. See the *C6000-to-C7000 Migration Users Guide* (SPRUIG5) for information about differences and migration.

1.1 Software Development Tools Overview....17
1.2 Compiler Interface....18
1.3 ANSI/ISO Standard....19
1.4 Output Files....19

Copyright © 2026 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.

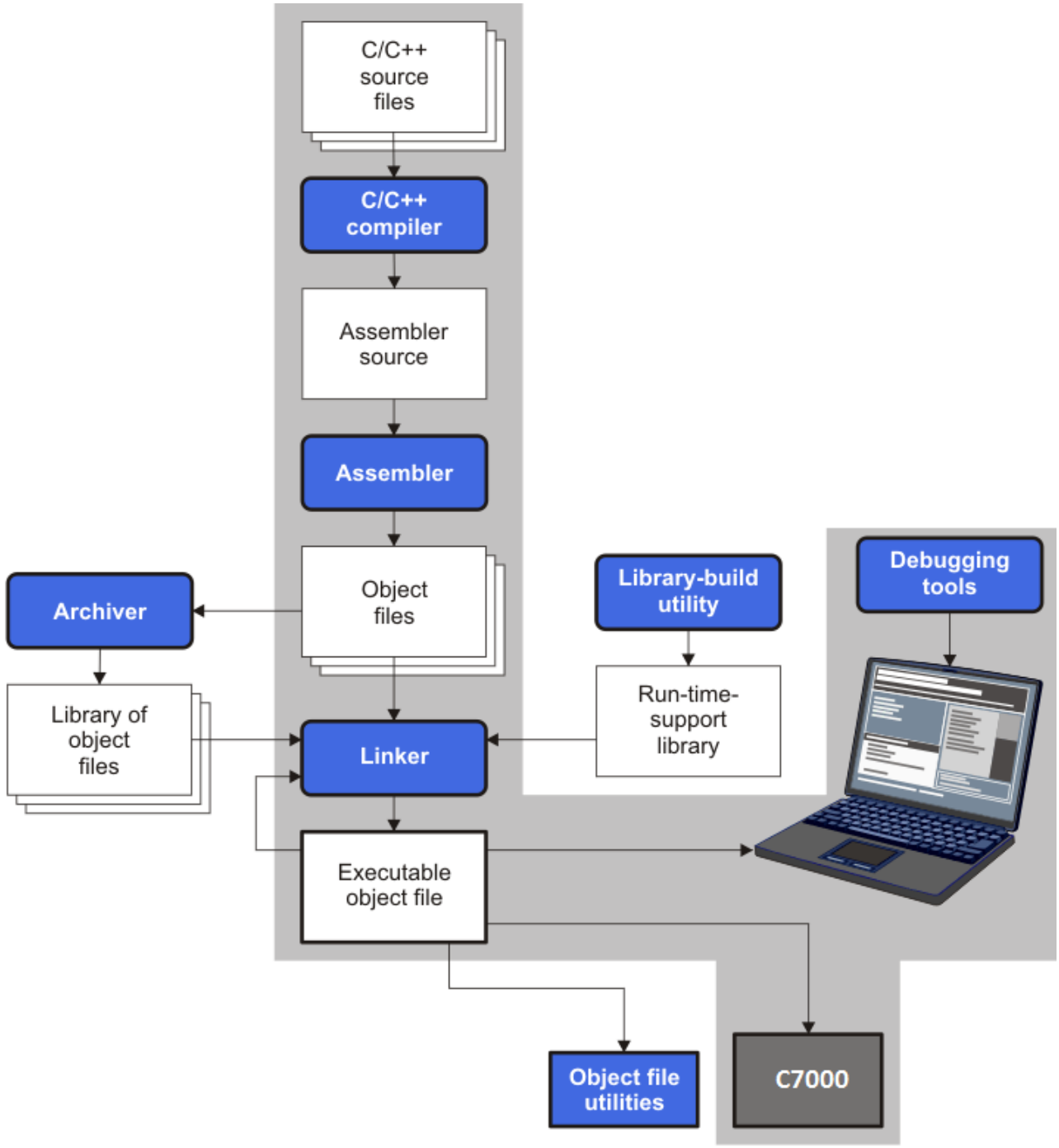


Figure 1-1. C7000 Software Development Flow

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

- The **compiler** accepts C/C++ source code and produces C7000 assembly language source code, which it automatically translates into machine language relocatable object files. See Chapter 3.

Copyright © 2026 Texas Instruments Incorporated

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

  The **library-build utility** automatically builds the run-time-support library if compiler and linker options require a custom version of the library. See Section 7.4. Source code for the standard run-time-support library functions for C and C++ is provided in the lib\src subdirectory of the directory where the compiler is installed.
- The **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 14.
- The main product of this development process is an executable object file that can be executed on a C7000 CPU that is part of a larger device.

In addition, the following utilities are provided to help examine or manage the content of a given object file:

- The **object file display utility** prints the contents of object files and object libraries in either human readable or XML formats. See Section 13.1.
- The **disassembler** decodes the machine code from object modules to show the assembly instructions that it represents. See Section 13.2.
- The **name utility** prints a list of symbol names for objects and functions defined or referenced in an object file or object archive. See Section 13.3.
- The **strip utility** removes symbol table and debugging information from object files and object libraries. See Section 13.4.

1.2 Compiler Interface

The compiler is a command-line program named cl7x. This program can compile, optimize, assemble, and link programs in a single step. Within Code Composer Studio™, the compiler is run automatically to perform the steps needed to build a project.

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

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

Copyright © 2026 Texas Instruments Incorporated

1.3 ANSI/ISO Standard

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

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

  The C language is also described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R).
- **ISO-standard C++:** The compiler uses the C++14 version of the C++ standard. See the C++ Standard ISO/IEC 14882:2014. For a description of *unsupported* C++ features, see Section 5.2.
- **ISO-standard run-time support:** The compiler tools come with an extensive run-time library. Library functions conform to the ISO C/C++ library standard unless otherwise stated. The library includes functions for standard input and output, string manipulation, dynamic memory allocation, data conversion, timekeeping, trigonometry, and exponential and hyperbolic functions. Functions for signal handling are not included, because these are target-system specific. For more information, see Chapter 7.

See Section 5.12 for command line options to select the C or C++ standard your code uses.

1.4 Output Files

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

- **ELF object files.** Executable and Linking Format (ELF) enables supporting modern language features like early template instantiation and exporting inline functions. ELF is part of the System V Application Binary Interface (ABI). The ELF format used for C7000 is extended by the C7000 Embedded Application Binary Interface (EABI).

Copyright © 2026 Texas Instruments Incorporated

Chapter 2

Getting Started with the Code Generation Tools

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

2.1 How Code Composer Studio Projects Use the Compiler 21
2.2 Compiling from the Command Line 21

Copyright © 2026 Texas Instruments Incorporated

2.1 How Code Composer Studio Projects Use the Compiler

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

Table 2-1. Steps for Creating a CCS Project

Step	Effects on Use of the Compiler
1. Choose **File > New > CCS Project** from the menus.	
2. In the New CCS Project wizard, first select the **Target**. You can use the drop-down on the left to filter the list of specific targets on the right.	Sets the --silicon_version (-mv) compiler option. See Section 3.3.5. In addition, a preprocessor symbol matching the target is defined using the --define compiler option. See Section 3.3.2.
3. In the **Connection** field, select the method you will use to connect to the device.	Generates a target configuration file for use when running the project.
4. In the **Project name** field, type a name for the project.	Determines the folder where the project is stored.
5. Expand the **Advanced settings** area.	
6. Make sure the **Compiler version** you want to use is selected.	Sets the --include_path compiler option to the include directory for that version of the Code Generation Tools. See Section 3.5.2.1.
7. By default, C7000 applications are compiled to be little-endian. In the **Device endianness** field, you can choose big-endian if needed.	Sets the --big_endian compiler option if the default is not used. See Section 3.3.4.
8. The linker command file and runtime support library are selected automatically based on your choices in the other fields.	
9. Expand the **Project templates and examples** area.	
10. Select a template for your project. The project templates you can choose from include a completely empty project with no source files, a project containing only main.c, and a Hello World example. Other examples that use plug-in software components you have installed are available in the TI Resource Explorer window.	
11. Click **Finish**.	

After you have created a CCS project, you can use the Properties dialog for the project to see how the compiler and linker will be used and modify the command-line options used when compiling and linking. To open this dialog, select the project in the Project Explorer and choose **Project > Properties** from the menus. Expand the category tree to select **Build > C7000 Compiler** and **Build > C7000 Linker**. You can learn more about any command-line options you see in this dialog in Chapter 3.

2.2 Compiling from the Command Line

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

The compiler and linker are run using the same executable. This executable is the **cl7x.exe** file, which is located in the **bin** subdirectory of your TI Code Generation Tools installation.

You can use a single command line to both compile your code to create object files and link the object files to create an executable. All the command-line options that occur before the --run_linker (or -z for short) option apply to the compiler. All the command-line options that occur after the --run_linker (-z) option apply to the linker. In the following command-line, the -mv7100, --c99, --opt_level, --define, and --include_path options are compiler options. The --library, --heap_size, and --output_file options are linker options.

```
cl7x -mv7100 --c99 --opt_level=1 --define=c7000 --include_path="C:/ti/ti-cgt-c7000/include"
    hello.c objects.cpp
    --run_linker --library=lnk.cmd --heap_size=0x800 --output_file=myprogram.out
```


Copyright © 2026 Texas Instruments Incorporated

Chapter 3

Using the C/C++ Compiler

The compiler translates your source program into machine language object code that the C7000 can execute. Source code must be compiled, assembled, and linked to create an executable file. All of these steps are executed at once by using the compiler.

3.1 About the Compiler 23
3.2 Invoking the C/C++ Compiler 23
3.3 Changing the Compiler's Behavior with Options 24
3.4 Controlling the Compiler Through Environment Variables 37
3.5 Controlling the Preprocessor 39
3.6 Passing Arguments to main() 44
3.7 Understanding Diagnostic Messages 44
3.8 Other Messages 47
3.9 Generating a Raw Listing File (--gen_preprocessor_listing Option) 47
3.10 Using Inline Function Expansion 48
3.11 Using Interlist 52
3.12 About the Application Binary Interface 53
3.13 Enabling Entry Hook and Exit Hook Functions 54

Copyright © 2026 Texas Instruments Incorporated

3.1 About the Compiler

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

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

  You can compile C and C++ files in a single command. The compiler uses the filename extensions to distinguish between different file types. See Section 3.3.10 for more information.
- The **linker** combines object files to create a static executable file. The link step is optional, so you can compile many modules independently and link them later. See Chapter 11 for information about linking the files.

Note

Invoking the Linker

By default, the compiler does not invoke the linker. You can invoke the linker by using the --run_linker (-z) compiler option. See Section 11.1.1 for details.

3.2 Invoking the C/C++ Compiler

To invoke the compiler, enter:

cl7x [*options*] [*filenames*] [**--run_linker** [*link_options*] *object files*]

cl7x	Command that runs the compiler.
options	Options that affect the way the compiler processes input files. The options are listed in Table 3-6 through Table 3-27.
filenames	One or more C/C++ source files.
--run_linker (**-z**)	Option that invokes the linker. The --run_linker option's short form is -z. See Chapter 11 for more information.
link_options	Options that control the linking process.
object files	Names of the object files for the linking process.

The arguments to the compiler are of three types:

- Compiler options
- Link options
- Filenames

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

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

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

```
cl7x symtab.c file.c --run_linker --library=lnk.cmd
       --output_file=myprogram.out
```


Copyright © 2026 Texas Instruments Incorporated

3.3 Changing the Compiler's Behavior with Options

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

For a help screen summary of the options, enter **cl7x** with no parameters on the command line.

The following apply to the compiler options:

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

You can define default options for the compiler by using the C7X_C_OPTION environment variable. For a detailed description of the environment variable, see Section 3.4.1.

Table 3-1 through Table 3-27 summarize all options (including link options). Use the references in the tables for more complete descriptions of the options.

Table 3-1. Processor Options

Option	Alias	Effect	Section
--silicon_version=*id*	-mv	Selects target version. Defaults to 7100.	Section 3.3.5
--silicon_errata_i2117		Generates code to work around the C7100 silicon errata i2117.	Section 3.3.4
--silicon_errata_i2376={on|off}		Generates code to work around the C7504 silicon errata i2376.	Section 3.3.4
--silicon_errata_i2528={on|off}		Generates code to work around C71xx/C75xx silicon errata i2528.	Section 3.3.4
--silicon_errata_i2532={on|off}		Generates code to work around silicon errata i2532. This option is on by default for C7100, C7120, C7504, C7524, C7604.	Section 3.3.4
--big_endian	-me	Produces object code in big-endian format. The default format is little-endian.	Section 3.3.4
--mma_version={ 1 | 2 | 2_256 | 2_256f | disallow }		Specify the version of the Matrix Multiply Accelerator (MMA) to be used on the target device.	Section 3.3.4

Table 3-2. Optimization Options(1)

Option	Alias	Effect	Section
--opt_level=off		Disables all optimization (default if option not used and --vectypes=off).	Section 4.1
--opt_level=*n*	-O*n*	Level 0 (-O0) optimizes register usage only (default if option not used and --vectypes=on). Level 1 (-O1) uses Level 0 optimizations and optimizes locally. Level 2 (-O2) uses Level 1 optimizations and optimizes globally (default if option used with no setting). Level 3 (-O3) uses Level 2 optimizations and optimizes the file. Level 4 (-O4) uses Level 3 optimizations and performs link-time optimization.	Section 4.1, Section 4.3
--opt_for_speed[=*n*]	-mf	Controls the tradeoff between size and speed (0-5 range). If this option is not specified or is specified without *n*, the default value is 4.	Section 4.2

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

Copyright © 2026 Texas Instruments Incorporated

Table 3-3. Advanced Optimization Options(1)

Option	Alias	Effect	Section
--assume_addresses_ok_for_stream={off|on}		If enabled, the automatic streaming pass, which is controlled with the --auto_stream option, assumes that addressing patterns map to a stream legally. Enabling this assumption can result in more use of a Streaming Engine (SE) or Streaming Address Generator (SA). Situations that could require such an assumption include, but are not limited to: 64-bit indexes, unsigned indexes, and array indexing applied to vector pointers.	Section 4.15.9
--auto_inline=[*size*]	-oi	Sets automatic inlining size (--opt_level=3 only). If *size* is not specified, the default is 1.	Section 4.5
--auto_stream={off|saving|no_saving}		Controls whether the compiler attempts to automatically use the Streaming Engine (SE) and Streaming Address Generator (SA) where possible and profitable. In "off" mode, automatic use of the SE and SA is disabled. In "saving" mode, the SE and/or SA state are saved in case they are already in use. In "no_saving" mode, automatic use of the SE and SA is enabled without context saving. C7100 and C7120 devices support only "off" or "no_saving" modes and "off" is the default. Newer devices, such as C7504, support all three modes and --auto_stream=saving is the default.	Section 4.15.9
--call_assumptions=*n*	-op*n*	Level 0 (-op0) specifies that the module contains functions and variables that are called or modified from outside the source code provided to the compiler. Level 1 (-op1) specifies that the module contains variables modified from outside the source code provided to the compiler but does not use functions called from outside the source code. Level 2 (-op2) specifies that the module contains no functions or variables that are called or modified from outside the source code provided to the compiler (default). Level 3 (-op3) specifies that the module contains functions that are called from outside the source code provided to the compiler but does not use variables modified from outside the source code.	Section 4.4.1
--disable_inlining		Prevents any inlining from occurring.	Section 3.10
--fp_mode={relaxed|strict}		Enables or disables relaxed floating-point mode.	Section 3.3.3
--fp_reassoc={on|off}		Enables or disables the reassociation of floating-point arithmetic.	Section 3.3.3
--fp_single_precision_constant		Causes all unsuffixed floating-point constants to be treated as single precision values instead of as double-precision constants.	Section 3.3.3
--gen_opt_info=*n*	-on*n*	Level 0 (-on0) disables the optimization information file. Level 1 (-on1) produces an optimization information file. Level 2 (-on2) produces a verbose optimization information file.	Section 4.3.1
--optimizer_interlist	-os	Interlists optimizer comments with assembly statements.	Section 4.12
--program_level_compile	-pm	Combines source files to perform program-level optimization.	Section 4.4
--src_interlist	-s	Interlists optimizer comments (if available) and assembly source statements; otherwise interlists C and assembly source statements.	Section 3.3.2
--aliased_variables	-ma	Notifies the compiler that addresses passed to functions may be modified by an alias in the called function.	Section 4.9.1

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

Copyright © 2026 Texas Instruments Incorporated

Table 3-4. Debug Options

Option	Alias	Effect	Section
--symdebug:dwarf	-g	Default behavior. Enables symbolic debugging. The generation of debug information does not impact optimization. Therefore, generating debug information is enabled by default.	Section 3.3.6 Section 4.13
--symdebug:dwarf_version=3|4		Specifies the DWARF format version. The default version is 4.	Section 3.3.6
--symdebug:none		Disables all symbolic debugging.	Section 3.3.6 Section 4.13

Table 3-5. Include Options

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

Table 3-6. Control Options

Option	Alias	Effect	Section
--compile_only	-c	Disables linking (negates --run_linker).	Section 11.1.3
--help	-h	Prints (on the standard output device) a description of the options understood by the compiler.	Section 3.3.2
--run_linker	-z	Causes the linker to be invoked from the compiler command line.	Section 3.3.2
--skip_assembler	-n	Compiles C/C++ source file , producing an assembly language output file. The assembler is not run and no object file is produced.	Section 3.3.2
--keep_asm	-k	Keeps the assembly language (.asm) file.	

Table 3-7. Language Options

Option	Alias	Effect	Section
--c89		Processes C files according to the ISO C89 standard.	Section 5.12
--c99		Processes C files according to the ISO C99 standard.	Section 5.12
--c11		Processes C files according to the ISO C11 standard.	Section 5.12
--c++14		Processes C++ files according to the ISO C++14 standard.	Section 5.12
--cpp_default	-fg	Processes all source files with a C extension as C++ source files.	Section 3.3.8
--deprecated_logical_vector_semantics		Enables use of legacy behavior for handling Boolean vectors, relational operators, and logical operators.	Section 5.14.8.4
--exceptions		Enables C++ exception handling.	Section 5.6
--extern_c_can_throw		Allows extern C functions to propagate exceptions.	--
--float_operations_allowed ={none|all|32|64}		Restricts the types of floating point operations allowed.	Section 3.3.3
--pending_instantiations=#		Specify the number of template instantiations that may be in progress at any given time. Use 0 to specify an unlimited number.	Section 3.3.4
--printf_support={nofloat|full| minimal}		Enables support for smaller, limited versions of the printf function family (sprintf, fprintf, etc.) and the scanf function family (sscanf, fscanf, etc.) run-time-support functions.	Section 3.3.3
--relaxed_ansi	-pr	Enables relaxed mode; ignores strict ISO violations. This is on by default. To disable this mode, use the --strict_ansi option.	Section 5.12.3
--strict_ansi	-ps	Enables strict ANSI/ISO mode (for C/C++, not for K&R C). In this mode, language extensions that conflict with ANSI/ISO C/C++ are disabled. In strict ANSI/ISO mode, most ANSI/ISO violations are reported as errors. Violations that are considered discretionary may be reported as warnings instead.	Section 5.12.3
--vectypes={on|off}		Enable support for TI vector data types. On by default.	Section 5.3.2

Copyright © 2026 Texas Instruments Incorporated

Table 3-8. Parser Preprocessing Options

Option	Alias	Effect	Section
--preproc_dependency[=*filename*]	-ppd	Performs preprocessing only, but instead of writing preprocessed output, writes a list of dependency lines suitable for input to a standard make utility.	Section 3.5.8
--preproc_includes[=*filename*]	-ppi	Performs preprocessing only, but instead of writing preprocessed output, writes a list of files included with the #include directive.	Section 3.5.9
--preproc_macros[=*filename*]	-ppm	Performs preprocessing only. Writes list of predefined and user-defined macros to a file with the same name as the input but with a .pp extension.	Section 3.5.10
--preproc_only	-ppo	Performs preprocessing only. Writes preprocessed output to a file with the same name as the input but with a .pp extension.	Section 3.5.4
--preproc_with_comment	-ppc	Performs preprocessing only. Writes preprocessed output, keeping the comments, to a file with the same name as the input but with a .pp extension.	Section 3.5.6
--preproc_with_compile	-ppa	Continues compilation after preprocessing with any of the -pp<x> options that normally disable compilation.	Section 3.5.5
--preproc_with_line	-ppl	Performs preprocessing only. Writes preprocessed output with line-control information (#line directives) to a file with the same name as the input but with a .pp extension.	Section 3.5.7

Table 3-9. Predefined Macro Options

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

Table 3-10. Diagnostic Message Options

Option	Alias	Effect	Section
--compiler_revision		Prints out the compiler release revision and exits.	--
--diag_error=*num*	-pdse	Categorizes the diagnostic identified by *num* as an error.	Section 3.7.1
--diag_remark=*num*	-pdsr	Categorizes the diagnostic identified by *num* as a remark.	Section 3.7.1
--diag_suppress=*num*	-pds	Suppresses the diagnostic identified by *num*.	Section 3.7.1
--diag_warning=*num*	-pdsw	Categorizes the diagnostic identified by *num* as a warning.	Section 3.7.1
--diag_wrap={on|off}		Wrap diagnostic messages (default is on). Note that this command-line option cannot be used within the Code Composer Studio IDE.	
--display_error_number	-pden	Displays a diagnostic's identifiers along with its text. Note that this command-line option cannot be used within the Code Composer Studio IDE.	Section 3.7.1
--emit_warnings_as_errors	-pdew	Treat warnings as errors.	Section 3.7.1
--issue_remarks	-pdr	Issues remarks (non-serious warnings).	Section 3.7.1
--no_warnings	-pdw	Suppresses diagnostic warnings (errors are still issued).	Section 3.7.1
--quiet	-q	Suppresses progress messages (quiet).	--
--set_error_limit=*num*	-pdel	Sets the error limit to *num*. The compiler abandons compiling after this number of errors. (The default is 100.)	Section 3.7.1
--super_quiet	-qq	Super quiet mode.	--
--tool_version	-version	Displays version number for each tool.	--
--verbose		Display banner and function progress information.	--
--verbose_diagnostics	-pdv	Provides verbose diagnostic messages that display the original source with line-wrap. Note that this command-line option cannot be used within the Code Composer Studio IDE.	Section 3.7.1
--write_diagnostics_file	-pdf	Generates a diagnostic message information file. Compiler only option. Note that this command-line option cannot be used within the Code Composer Studio IDE.	Section 3.7.1

Copyright © 2026 Texas Instruments Incorporated

Table 3-11. Supplemental Information Options

Option	Alias	Effect	Section
--gen_preprocessor_listing	-pl	Generates a raw listing file (.rl).	Section 3.9
--section_sizes={on|off}		Generates section size information, including sizes for sections containing executable code and constants, constant or initialized data (global and static variables), and uninitialized data. (Default is off if this option is not included on the command line. Default is on if this option is used with no value specified.)	Section 3.7.1

Table 3-12. Run-Time Model Options

Option	Alias	Effect	Section
--common={on|off}		On by default. When on, uninitialized file scope variables are emitted as common symbols. When off, common symbols are not created.	Section 3.3.4
--debug_software_pipeline	-mw	Produce verbose software pipelining report.	Section 4.7.2
--disable_software_pipeline	-mu	Turns off software pipelining.	Section 4.7.1
--fp_not_associative	-mc	Prevents reordering of associative floating-point operations.	Section 4.10
--gen_data_subsections={on|off}		Place all aggregate data (arrays, structs, and unions) into subsections. This gives the linker more control over removing unused data during the final link step. See the link to the right for details about the default setting.	Section 11.2.3
--gen_func_subsections={on|off}	-mo	Puts each function in a separate subsection in the object file. If this option is not used, the default is off. See the link to the right for details about the default setting.	Section 11.2.2
--ramfunc={on|off}		If set to on, indicates that each function will be run from RAM. Functions will be placed in RAM and optimized for RAM execution. Equivalent to specifying __attribute__((ram_func)) on all functions in the translation unit.	Section 3.3.4

Table 3-13. Entry/Exit Hook Options

Option	Alias	Effect	Section
--entry_hook[=*name*]		Enables entry hooks.	Section 3.13
--entry_parm={none|name|address}		Specifies the parameters to the function to the --entry_hook option.	Section 3.13
--exit_hook[=*name*]		Enables exit hooks.	Section 3.13
--exit_parm={none|name|address}		Specifies the parameters to the function to the --exit_hook option.	Section 3.13
--remove_hooks_when_inlining		Removes entry/exit hooks for auto-inlined functions.	Section 3.13

Table 3-14. Assembly Options

Option	Alias	Effect	Section
--c_src_interlist	-ss	Interlists C source and assembly statements.	Section 3.11 Section 4.12

Table 3-15. 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 3.3.8
--c_file=*filename*	-fc	Identifies *filename* as a C source file regardless of its extension. By default, the compiler treats .c files as C source files.	Section 3.3.8
--cpp_file=*filename*	-fp	Identifies *filename* as a C++ file, regardless of its extension. By default, the compiler treats .C, .cpp, .cc and .cxx files as C++ files.	Section 3.3.8
--obj_file=*filename*	-fo	Identifies *filename* as an object code file regardless of its extension. By default, the compiler and linker treat .obj files as object code files, including both *.c.obj and *.cpp.obj files.	Section 3.3.8

Copyright © 2026 Texas Instruments Incorporated

Table 3-16. Directory Specifier Options

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

Table 3-17. Default File Extensions Options

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

Table 3-18. Command Files Options

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

Table 3-19. Performance Advisor Options

Option	Alias	Effect	Section
--advice:performance[={all|none}]		Generates compiler optimization advice. Default is all.	Section 4.11
--advice:performance_file={stdout|stderr|user_specified_filename}		Specifies that advice be written to stdout, stderr, or a file.	Section 4.11
--advice:performance_dir={user_specified_directory_name}		Specifies that advice file be created in the named directory.	Section 4.11

Copyright © 2026 Texas Instruments Incorporated

3.3.1 Linker Options

The following tables list the linker options. See Chapter 11 of this document and Chapter 12 for details on these options.

Table 3-20. Linker Basic Options

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

Table 3-21. File Search Path Options

Option	Alias	Description
--library=*file*	-l	Names an archive library or link command *file* as linker input.
--disable_auto_rts		Disables the automatic selection of a run-time-support library. See Section 11.3.1.1.
--priority	-priority	Satisfies unresolved references by the first library that contains a definition for that symbol.
--reread_libs	-x	Forces rereading of libraries, which resolves back references.
--search_path=*pathname*	`-I`	Alters library-search algorithms to look in a directory named with *pathname* before looking in the default location. This option must appear before the --library option.
--start-group --end-group		Create a group of libraries to be used together to resolve references.

Table 3-22. Command File Preprocessing Options

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

Table 3-23. Diagnostic Message Options

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

Copyright © 2026 Texas Instruments Incorporated

Table 3-24. Linker Output Options

Option	Alias	Description
--absolute_exe	-a	Produce an absolute, executable object file. This is the default; if neither --absolute_exe nor --relocatable is specified, the linker acts as if --absolute_exe is specified.
--mapfile_contents=*attribute*		Controls the information that appears in the map file.
--relocatable	-r	Produces a nonexecutable, relocatable output object file.
--xml_link_info=*file*		Generate an XML *file* containing detailed information about the result of a link.

Table 3-25. Symbol Management Options

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

Table 3-26. Run-Time Environment Options

Option	Alias	Description
--arg_size=*size*	--args	Reserve *size* bytes for the argc/argv memory area.
--cinit_compression[=type]		Specifies the type of compression to apply to the C auto initialization data. The default if this option is specified with no *type* is lzss for Lempel-Ziv-Storer-Szymanski compression.
--copy_compression[=type]		Compresses data copied by linker copy tables. The default if this option is specified with no *type* is lzss for Lempel-Ziv-Storer-Szymanski compression.
--fill_value=*value*	-f	Sets default fill value for holes within output sections
--honor_cmdfile_order		Place output sections in the order listed in the linker command file.
--ram_model	-cr	Initializes variables at load time. See Section 11.3.4 for details.
--rom_model	-c	Autoinitializes variables at run time. See Section 11.3.4 for details.
--trampolines[=off|on]		Generates far call trampolines. Default is on.

Table 3-27. Miscellaneous Options

Option	Alias	Description
--compress_dwarf[=off|on]		Aggressively reduces size of DWARF information from input object files. Default is on.
--linker_help	[-]-help	Displays information about syntax and available options.
--minimize_trampolines[=off | postorder]		Places sections to minimize number of far trampolines required. Default is postorder.
--preferred_order=*function*		Prioritizes placement of functions.
--zero_init=[off|on]		Controls preinitialization of uninitialized variables. Default is on. Always off if --ram_model is used.

Copyright © 2026 Texas Instruments Incorporated

3.3.2 Frequently Used Options

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

--c_src_interlist Invokes the interlist feature, which interweaves original C/C++ source with compiler-generated assembly language. The interlisted C statements may appear to be out of sequence. You can use the interlist feature with the optimizer by combining the --optimizer_interlist and --c_src_interlist options. See Section 4.12. The --c_src_interlist option can have a negative performance and/or code size impact.

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

```
cl7x --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 C7X_C_OPTION environment variable and you do not want to link. See Section 11.1.3.

--define=*name[=def]* Predefines the constant *name* for the preprocessor. This is equivalent to inserting #define *name def* at the top of each C source file. If the optional[=*def*] is omitted, *name* is set to 1. This option's short form is -D.

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

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

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

--include_path=*directory* Adds *directory* to the list of directories that the compiler searches for #include files. The --include_path option's short form is `-I`. You can use this option several times to define several directories; be sure to separate the --include_path options with spaces. If you do not specify a directory name, the preprocessor ignores the --include_path option. See Section 3.5.2.1.

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

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

--run_linker Runs the linker on the specified object files. The --run_linker option and its parameters follow all other options on the command line. All arguments that follow --run_linker are passed to the linker. The --run_linker option's short form is -z. See Section 11.1.

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

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

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

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

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

Copyright © 2026 Texas Instruments Incorporated

3.3.3 Miscellaneous Useful Options

Following are detailed descriptions of miscellaneous options:

--advice:performance Generates compile-time optimization advice. See Section 4.11.

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

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

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

In the following examples, iN=integer variable, fN=float variable, and dN=double variable:

```
il = f1 + f2 * 5.0 -> +, * are float, 5.0 is converted to 5.0f
il = d1 + d2 * d3 -> +, * are float
f1 = f2 + f3 * 1.1; -> +, * are float, 1.1 is converted to 1.1f
```

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

If --strict_ansi is specified, --fp_mode=strict is set automatically. You can enable the relaxed floating-point mode with strict ANSI mode by specifying --fp_mode=relaxed after --strict_ansi.

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

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

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

--preinclude=*filename* Includes the source code of *filename* at the beginning of the compilation. This can be used to establish standard macro definitions. The filename is searched for in the directories on the include search list. The files are processed in the order in which they were specified.

--printf_support={full|nofloat|minimal} Enables support for smaller, limited versions of the printf function family (sprintf, fprintf, etc.) and the scanf function family (sscanf, fscanf, etc.) run-time-support functions. The valid values are:

- full: Supports all format specifiers. This is the default.
- nofloat: Excludes support for printing and scanning floating-point values. Supports all format specifiers except %a, %A, %f, %F, %g, %G, %e, and %E.
- minimal: Supports the printing and scanning of integer, char, or string values without width or precision flags. Specifically, only the %%, %d, %o, %c, %s, and %x format specifiers are supported.

There is no run-time error checking to detect if a format specifier is used for which support is not included. The --printf_support option precedes the --run_linker option, and must be used when performing the final link.

Copyright © 2026 Texas Instruments Incorporated

3.3.4 Run-Time Model Options

These options are specific to the C7000 toolset. See the referenced sections for more information.

--big_endian	Produces code in big-endian format. By default, little-endian code is produced.
--common={on|off}	When on (the default), uninitialized file scope variables are emitted as common symbols. When off, common symbols are not created. The benefit of allowing common symbols to be created is that generated code can remove unused variables that would otherwise increase the size of the .bss section. (Uninitialized variables of a size larger than 32 bytes are separately optimized through placement in separate subsections that can be omitted from a link.) Variables cannot be common symbols if they are assigned to a section other than .bss or have a specified memory bank.
--debug_software_pipeline	Produces verbose software pipelining report. See Section 4.7.2.
--disable_software_pipeline	Turns off software pipelining. See Section 4.7.1.
--fp_not_associative	Compiler does not reorder floating-point operations. See Section 4.10.
--mma_version={ 1 | 2 | 2_256 | 2_256f | 3_1024 | disallow }	Specifies the Matrix Multiply Accelerator (MMA) revision on the device to be used. For a specific C7000 silicon version, the setting of --mma_version may be the MMA version that corresponds to the silicon version (which is the default) or `disallow`. Specifying an MMA version that does not match the silicon version is prohibited and causes an error. • For --silicon_version=7100, the default is --mma_version=1. • For --silicon_version=7120, the default is --mma_version=2. • For --silicon_version=7504, the default is --mma_version=2_256. • For --silicon_version=7524, the default is --mma_version=2_256 and 2_256f is supported. • For --silicon_version=7604, the default is --mma_version=3_1024. The compiler places appropriate MMA build attributes in object files it generates. Object files specify that the MMA is not used (UNUSED) if either the --mma_version=disallow option was used to disable use of the MMA or if the code in the application performs no actions that could make use of the MMA. MMA build attributes ensure that linking is disallowed for object files that use incompatible versions of the MMA. Versions 1, 2, and 2_256 of the MMA are all incompatible with each other. Object files generated with the 2_256 and 2_256f versions are compatible with each other. Object files with the MMA version set to UNUSED may be linked with files that use any MMA version (but not with incompatible versions).
--pending_instantiations=#	Specify the number of template instantiations that may be in progress at any given time. Use 0 to specify an unlimited number.
--ramfunc={on|off}	If set to on, indicates that each function will be run from RAM. Functions will be placed in RAM and optimized for RAM execution. Equivalent to specifying __attribute__((ram_func)) on all functions in the translation unit. If set to off, only functions with the ramfunc function attribute are treated this way. See Section 5.13.2.
--silicon_version=*num*	Selects the target CPU version. See Section 3.3.5.
--silicon_errata_i2117	The --silicon_errata_i2117 compiler option generates code that automatically works around silicon errata i2117 on devices with the C7100 CPU core. MMA performance may be negatively impacted by the use of this option in edge cases. This option is off by default.
--silicon_errata_i2376={on|off}	The --silicon_errata_i2376=on compiler option generates code that automatically works around silicon errata i2376 on devices with the C7504 CPU core. Performance should not be significantly affected by this workaround. This option is on by default if the --silicon_version=7504 compiler option (or the -mv7504 alias) is used. To turn off use of this workaround, use --silicon_errata_i2376=off. Turning off this workaround is not recommended; it is only intended to be turned off by advanced users in specific situations.
--silicon_errata_i2528={on|off}	The --silicon_errata_i2528=on compiler option generates code that automatically works around the silicon errata i2528 defect on devices with C71xx or C75xx CPU cores. This defect caused misclassification of the load/store category of an instruction when checking memory permissions. When on, this option impacts performance, typically by 3-5% on average or by as much as 50% in isolated cases. This option is on by default if the --silicon_version compiler option (or the -mv alias) is set to 7100, 7120, 7504, or 7524. The defect is fixed for 76xx and later devices. To turn off use of this workaround, use --silicon_errata_i2528=off. This workaround may be safely turned off if all MMU pages are configured to be writable.

Copyright © 2026 Texas Instruments Incorporated

--silicon_errata_i2532={on|off}	The --silicon_errata_i2532=on compiler option generates code that automatically works around the silicon errata i2532 defect. This option is on by default for C7100, C7120, C7504, C7524, and C7604 devices. In rare cases, this defect causes instruction predicates with SAOPEN to receive a stale instruction predicate value. The compiler works around this issue by disallowing the use of instruction predicates with SAOPEN. This option typically has no performance impact for the vast majority of code; the worst case observed was a 2% impact.

3.3.5 Selecting Target CPU Version (--silicon_version Option)

The --silicon_version option controls the use of target-specific instructions and alignment. The alias for this option is -mv. If this option is not used, the compiler generates code for the C7100 parts by default. Currently, the available settings for this option are:

- --silicon_version=7100 (or its alias, -mv7100)
- --silicon_version=7120 (or its alias, -mv7120)
- --silicon_version=7504 (or its alias, -mv7504)
- --silicon_version=7524 (or its alias, -mv7524)
- --silicon_version=7604 (or its alias, -mv7604)

3.3.6 Symbolic Debugging and Profiling Options

The following options are used to select symbolic debugging :

--symdebug:dwarf	(Default) Generates directives that are used by the C/C++ source-level debugger . The --symdebug:dwarf option's short form is -g. See Section 4.13. For details on the DWARF format, see *The DWARF Debugging Standard*.
--symdebug:dwarf_version={3|4}	Specifies the DWARF debugging format version (3 or 4) to be generated when --symdebug:dwarf (the default) is specified. By default, the compiler generates DWARF version 4 debug information. For more information on TI extensions to the DWARF language, see *The Impact of DWARF on TI Object Files* (SPRAAB5).
--symdebug:none	Disables all symbolic debugging output. This option is not recommended; it prevents debugging and most performance analysis capabilities.

3.3.7 Specifying Filenames

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

Extension	File Type
.c	C source
.C	Depends on operating system
.cpp, .cxx, .cc	C++ source
.obj .c.obj .cpp.obj .o* .dll .so	Object

Note

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

For information about how you can alter the way that the compiler interprets individual filenames, see Section 3.3.8. For information about how you can alter the way that the compiler interprets and names the extensions of files, see Section 3.3.11.

Copyright © 2026 Texas Instruments Incorporated

You can use wildcard characters to compile 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:

```
cl7x *.cpp
```

Note

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

3.3.8 Changing How the Compiler Interprets Filenames

You can use options to change how the compiler interprets your filenames. If the extensions that you use are different from those recognized by the compiler, you can use the filename options to specify the type of file. You can insert an optional space between the option and the filename. Select the appropriate option for the type of file you want to specify:

--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 a C++ file called objects.cp, use the --cpp_file and --c_file options to force the correct interpretation:

```
cl7x --c_file=file.s --cpp_file=objects.cp
```

You cannot use the filename options with wildcard specifications.

Note

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

3.3.9 Changing How the Compiler Processes C Files

The --cpp_default option causes the compiler to process C files as C++ files. By default, the compiler treats files with a .c extension as C files. See Section 3.3.10 for more information about filename extension conventions.

3.3.10 Changing How the Compiler Interprets and Names Extensions

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

--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

Copyright © 2026 Texas Instruments Incorporated

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

```
cl7x --cpp_extension=.rrr --obj_extension=.o fit.rrr
```

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

```
cl7x --cpp_extension=rrr --obj_extension=o fit.rrr
```

3.3.11 Specifying Directories

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

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

```
cl7x --asm_directory=d:\assembly
```

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

```
cl7x --obj_directory=d:\object
```

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

```
cl7x --output_file=transfer
```

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

```
cl7x --pp_directory=d:\preproc
```

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

```
cl7x --temp_directory=d:\temp
```

Note

Specifying a Location for Temporary Files: The effects of the --temp_directory (-ft for short) option are limited. It is used only if the default directory is inaccessible (for example, because you are compiling from a read-only directory). Also, it only controls placement of certain temporary files that are usually deleted before the compiler exits; it does not control placement of compiler-generated assembly files or object files.

To change the directory where all temporary files created by the compiler are placed, set the environment variable TMP before running the compiler on Windows, or set TMPDIR before running the compiler on other host systems. See Section 3.4.3 for details.

3.4 Controlling the Compiler Through Environment Variables

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

Note

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

3.4.1 Setting Default Compiler Options (C7X_C_OPTION)

You might find it useful to set the compiler and linker default options using the C7X_C_OPTION environment variable. If you do this, the compiler and linker use the default options and/or input filenames in the C7X_C_OPTION definition every time you run them.

Setting default options with the environment variable is useful if 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 input filenames, it looks for the C7X_C_OPTION environment variable and processes it. The table below shows how to set the C7X_C_OPTION environment variable. Select the command for your operating system:

Copyright © 2026 Texas Instruments Incorporated

Operating System	Enter
UNIX (Bourne shell)	**C7X_C_OPTION="** *option*$_1$ [*option*$_2$ . . .]**"; export C7X_C_OPTION**
Windows	**set C7X_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 C7X_C_OPTION environment variable as follows:

```
set C7X_C_OPTION=--quiet --src_interlist --run_linker
```

Any options following --run_linker on the command line or in C7X_C_OPTION are passed to the linker. Thus, you can use the C7X_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 C7X_C_OPTION is set as shown above:

```
cl7x *c                          ; compiles and links
cl7x --compile_only *.c          ; only compiles
cl7x *.c --run_linker lnk.cmd    ; compiles and links using a command file
cl7x --compile_only *.c --run_linker lnk.cmd ; only compiles (--compile_only overrides --run_linker)
```

For details on compiler options, see Section 3.3. For details on linker options, see Section 12.4.

3.4.2 Naming One or More Alternate Directories (C7X_C_DIR)

The linker uses the C7X_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)	**C7X_C_DIR="** *pathname*$_1$ **;***pathname*$_2$ **;..."; export C7X_C_DIR**
Windows	**set C7X_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 C7X_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 C7X_C_DIR=c:\first path\to\tools;d:\second path\to\tools
```

The environment variable remains set until you reboot the system or reset the variable by entering:

Operating System	Enter
UNIX (Bourne shell)	`unset C7X_C_DIR`
Windows	`set C7X_C_DIR=`

Copyright © 2026 Texas Instruments Incorporated

3.4.3 Controlling the Location of Temporary Files

The compiler places temporary files in the location identified by the appropriate environment variable for your operating system. If you have not set this environment variable, the compiler places temporary files following the operating system default.

- On Windows, set the TMP environment variable before running the compiler.
- On all other operating systems, set the TMPDIR environment variable before running the compiler.

You can use the --temp_directory (-ft for short) option to place certain temporary files, but the option's effects are limited. This option is used only if the default directory is inaccessible (for example, because you are compiling from a read-only directory). Also, this option controls placement only for temporary files that are usually deleted before the compiler exits; this option does not control placement of compiler-generated assembly files or object files.

3.5 Controlling the Preprocessor

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

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

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

3.5.1 Predefined Macro Names

The compiler maintains and recognizes the predefined macro names listed in Table 3-28.

Table 3-28. Predefined C7000 Macro Names

Macro Name	Description
__big_endian__	Defined to 1 if big-endian mode is selected (the --big_endian option is used); otherwise, it is undefined.
__C7000__	Defined to 1 for all C7000 subtargets.
__C7100__	Defined to 1 for the C7100 subtarget.
__C7120__	Defined to 1 for the C7120 subtarget.
__C7504__	Defined to 1 for the C7504 subtarget.
__C7524__	Defined to 1 for the C7524 subtarget.
__C7604__	Defined to 1 for the C7604 subtarget.
__C7X_MMA__	Defined to 1 unless the --mma_version=disallow option is specified.
__C7X_MMA_1__	Defined to 1 if the --mma_version=1 option is specified.
__C7X_MMA_2__	Defined to 1 if the --mma_version=2 option is specified.
__C7X_MMA_2_256__	Defined to 1 if the --mma_version=2_256 option is specified.
__C7X_MMA_2_256F__	Defined to 1 if the --mma_version=2_256f option is specified.
__C7X_MMA_3_1024__	Defined to 1 if the --mma_version=3_1024 option is specified.
__C7X_NUM_SA__	Defined to the number of Streaming Address Generators that are available. Currently always 4.
__C7X_NUM_SE__	Defined to the number of Streaming Engines that are available. Currently always 2.
__C7X_VEC_SIZE_BITS__	Defined to 1024, 512, or 256, depending on the --silicon_version/-mv option.
__C7X_VEC_SIZE_BYTES__	Defined to 128, 64, or 32, depending on the --silicon_version/-mv option.
__DATE__ (1)	Expands to the compilation date in the form *mmm dd yyyy*
__FILE__ (1)	Expands to the current source filename
_INLINE	Expands to 1 if optimization is used (--opt_level or -O option); undefined otherwise.

Copyright © 2026 Texas Instruments Incorporated

Table 3-28. Predefined C7000 Macro Names (continued)

Macro Name	Description
__LINE__ (1)	Expands to the current line number
__little_endian__	Defined to 1 if little-endian mode is selected (the --big_endian option is not used); otherwise, it is undefined.
__PTRDIFF_T_TYPE__	Defined to the type of ptrdiff_t, which is long.
__SIZE_T_TYPE__	Defined to the type of size_t, which is unsigned long.
__STDC__ (1)	Defined to 1 to indicate that compiler conforms to ISO C Standard. See Section 5.1 for exceptions to ISO C conformance.
__STDC_VERSION__	C standard macro.
__STDC_HOSTED__	C standard macro. Always defined to 1.
__STDC_NO_THREADS__	C standard macro. Always defined to 1.
__TI_C99_COMPLEX_ENABLED__	Defined to 1 if complex data types are enabled. This is always the case, though math operations are available only if complex.h is included.
__TI_COMPILER_VERSION__	Defined to a 7-9 digit integer, depending on if X has 1, 2, or 3 digits. The number does not contain a decimal. For example, version 3.2.1 is represented as 3002001. The leading zeros are dropped to prevent the number being interpreted as an octal.
__TI_EABI__	Always defined to 1.
__TI_GNU_ATTRIBUTE_SUPPORT__	Defined to 1 if GCC extensions are enabled (which is the default)
__TI_STRICT_ANSI_MODE__	Defined to 1 if strict ANSI/ISO mode is enabled (--strict_ansi is used); otherwise, defined as 0.
__TI_STRICT_FP_MODE__	Defined to 1 if --fp_mode=strict is used (default); otherwise, it is defined as 0.
__TIME__ (1)	Expands to the compilation time in the form "*hh:mm:ss*"
__WCHAR_T_TYPE__	Defined to the type of wchar_t, which is unsigned int.

(1) Specified by the ISO standard

You can use the names listed in Table 3-28 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");
```


Copyright © 2026 Texas Instruments Incorporated

3.5.2 The Search Path for #include Files

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

- If you enclose the filename in double quotes (" "), the compiler searches for the file in this order:
  1. The directory of the file that contains the #include directive and in the directories of any files that contain that file.
  2. Directories named with the --include_path option.
  3. Directories set with the C7X_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 C7X_C_DIR environment variable.

See Section 3.5.2.1 for information on using the --include_path option. See Section 3.4.2 for more information on input file directories.

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

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

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

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

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

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

Assume that the complete pathname for alt.h is:

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

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

Operating System	Enter
UNIX	`cl7x --include_path=/tools/files source.c`
Windows	`cl7x --include_path=c:\tools\files source.c`

Copyright © 2026 Texas Instruments Incorporated

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 C7X_C_DIR environment variable.

For example, if you set up C7X_C_DIR with the following command:

```
C7X_C_DIR "/usr/include;/usr/ucb"; export C7X_C_DIR
```

or invoke the compiler with the following command:

```
cl7x --include_path=/usr/include file.c
```

and file.c contains this line:

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

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

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

3.5.3 Support for the #warning and #warn Directives

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

If the --relaxed_ansi option is used (the default), both #warn and #warning preprocessor directives are supported.

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

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

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

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

3.5.5 Continuing Compilation After Preprocessing (--preproc_with_compile Option)

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

Copyright © 2026 Texas Instruments Incorporated

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

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

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

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

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

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

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

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

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

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

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

Copyright © 2026 Texas Instruments Incorporated

3.6 Passing Arguments to main()

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

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

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

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

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

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

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

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

For some RTOS-based executables, the elements in the .args section are as follows:

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

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

3.7 Understanding Diagnostic Messages

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

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

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

Copyright © 2026 Texas Instruments Incorporated

Diagnostic messages have a severity, as follows:

- A **fatal error** indicates a problem so severe that the compilation cannot continue. Examples of such problems include command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one will not be compiled.
- An **error** indicates a violation of the syntax or semantic rules of the C/C++ language. Compilation may continue, but object code is not generated.
- A **warning** indicates something that is likely to be a problem, but cannot be proven to be an error. For example, the compiler emits a warning for an unused variable. An unused variable does not affect program execution, but its existence suggests that you might have meant to use it. Compilation continues and object code is generated (if no errors are detected).
- A **remark** is less serious than a warning. It may indicate something that is a potential problem in rare cases, or the remark may be strictly informational. Compilation continues and object code is generated (if no errors are detected). By default, remarks are not issued. Use the --issue_remarks compiler option to enable remarks.
- An **advice** diagnostic indicates a potential way to improve your code. See Section 4.11 for details.

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

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

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

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

Long messages are wrapped to additional lines, when necessary.

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

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

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


Copyright © 2026 Texas Instruments Incorporated

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

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

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

3.7.1 Controlling Diagnostic Messages

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

--advice:performance By default, the compiler emits advice to stdout. Compile with **--advice:performance=none** to disable generation of performance advice. See Section 4.11 for related options and examples.

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

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

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

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

--display_error_number Displays a diagnostic's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (--diag_suppress, --diag_error, --diag_remark, and --diag_warning). This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix -D; otherwise, no suffix is present. See Section 3.7.

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

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

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

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

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

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

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

Copyright © 2026 Texas Instruments Incorporated

3.7.2 How You Can Use Diagnostic Suppression Options

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

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

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

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

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

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

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

Note

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

3.8 Other Messages

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

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

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

The raw listing file contains the following information:

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

Copyright © 2026 Texas Instruments Incorporated

Each source line in the raw listing file begins with one of the identifiers listed in Table 3-29.

Table 3-29. Raw Listing File Identifiers

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

The --gen_preprocessor_listing option also includes diagnostic identifiers as defined in Table 3-30.

Table 3-30. 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 numbercolumn number diagnostic

S	One of the identifiers in Table 3-30 that indicates the severity of the diagnostic
filename	The source file
line number	The line number in the source file
column number	The column number in the source file
diagnostic	The message text for the diagnostic

Diagnostic messages after the end of file are indicated as the last line of the file with a column number of 0. When diagnostic message text requires more than one line, each subsequent line contains the same file, line, and column information but uses a lowercase version of the diagnostic identifier. For more information about diagnostic messages, see Section 3.7.

3.10 Using Inline Function Expansion

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

Note

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

Function inlining is triggered by the following situations:

Copyright © 2026 Texas Instruments Incorporated

- The use of built-in intrinsic operations. Intrinsic operations look like function calls, and are inlined automatically, even though no function body exists.
- Use of the `inline` keyword or the equivalent `__inline` keyword. Functions declared with the inline keyword may be inlined by the compiler if you set --opt_level=0 or greater. The inline keyword is a suggestion from the programmer to the compiler. Even if your optimization level is high, inlining is still optional for the compiler. The compiler decides whether to inline a function based on the length of the function, the number of times it is called, your --opt_for_speed setting, and any contents of the function that disqualify it from inlining (see Section 3.10.2). Functions can be inlined at --opt_level=0 or above if the function body is visible in the same module or if -pm is also used and the function is visible in one of the modules being compiled. Functions may be inlined at link time if the file containing the definition and the call site were both compiled with --opt_level=4. Functions defined as both static and inline are more likely to be inlined.
- When --opt_level=3 or greater is used, the compiler may automatically inline eligible functions even if they are not declared as inline functions. The same list of decision factors listed for functions explicitly defined with the inline keyword is used. For more about automatic function inlining, see Section 4.5.
- The pragma FUNC_ALWAYS_INLINE (Section 5.8.12) and the equivalent `always_inline` attribute (Section 5.13.2) force a function to be inlined (where it is legal to do so) unless --opt_level=off. That is, the pragma FUNC_ALWAYS_INLINE forces function inlining even if the function is not declared as inline and the --opt_level=0 or --opt_level=1.
- The FORCEINLINE pragma (Section 5.8.10) forces functions to be inlined in the annotated statement. That is, it has no effect on those functions in general, only on function calls in a single statement. The FORCEINLINE_RECURSIVE pragma forces inlining not only of calls visible in the statement, but also in the inlined bodies of calls from that statement.
- The --disable_inlining option prevents any inlining. The pragma FUNC_CANNOT_INLINE prevents a function from being inlined. The NOINLINE pragma prevents calls within a single statement from being inlined. (NOINLINE is the inverse of the FORCEINLINE pragma.)

Note

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

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

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

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

3.10.1 Inlining Intrinsic Operators

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

Intrinsics are inlined whether or not you use the optimizer.

For details about intrinsics, and a list of the intrinsics, see Section 5.15. In addition to those listed, abs and memcpy are implemented as intrinsics.

Copyright © 2026 Texas Instruments Incorporated

3.10.2 Inlining Restrictions

The compiler makes decisions about which functions to inline based on the factors mentioned in Section 3.10. In addition, there are several restrictions that can disqualify a function from being inlined by automatic inlining or inline keyword-based inlining.

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

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

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

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

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

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

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

3.10.3 Unguarded Definition-Controlled Inlining

The inline keyword causes a function to be expanded inline at the point where it is called rather than using standard calling procedures. The compiler performs inline expansion of functions declared with the inline keyword.

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

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

Example 3-1. Using the Inline Keyword

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


Copyright © 2026 Texas Instruments Incorporated

3.10.4 Guarded Inlining and the _INLINE Preprocessor Symbol

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

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

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

In the following examples there are two definitions of the strlen function. The first (Example 3-2), in the header file, is an inline definition. This definition is enabled and the prototype is declared as static inline only if _INLINE is true (_INLINE is automatically defined for you when the optimizer is used).

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

Example 3-2. Header File string.h

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

Example 3-3. Library Definition File

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


Copyright © 2026 Texas Instruments Incorporated

3.11 Using Interlist

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

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

```
cl7x --c_src_interlist function
```

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

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

For information about using the interlist feature with the optimizer, see Section 4.12. Using the --c_src_interlist option can cause performance and/or code size degradation.

The C code for the foo() function in foo.c:

```
int foo(int a, int b, int
c)

{
    int d = a + b;
    int e = d - c;
    return e;
}
```

Is compiled with the following command:

```
cl7x foo.c --c_src_interlist --symdebug:none
```


Copyright © 2026 Texas Instruments Incorporated

Generates an assembly file, foo.asm, that contains in part:

```
;------------------------------------------------------------------------
;   1 | int foo(int a, int b, int c)
;------------------------------------------------------------------------

;******************************************************************************
;* FUNCTION NAME: foo                                                         *
;*                                                                            *
;*   Regs Modified     : A4,A8,D0,SP                                          *
;*   Regs Used         : A4,A5,A6,A8,D0,SP                                    *
;*   Local Frame Size  : 0 Args + 0 Auto + 8 Save = 8 byte                    *
;******************************************************************************
||foo||:
;** --------------------------------------------------------------------------*

           MVC     .S1     RP,A8             ; [A_S1]
||         STD     .D1     A8,*SP(8)         ; [A_D1]

           ADDD    .D1     SP,0xfffffff8,SP  ; [A_D1]
;------------------------------------------------------------------------
;   3 | int d = a + b;
;------------------------------------------------------------------------
           ADDW    .D1     A5,A4,D0          ; [A_D1] |3|
;------------------------------------------------------------------------
;   4 | int e = d - c;
;------------------------------------------------------------------------
           SUBW    .D1     D0,A6,A4          ; [A_D1] |4|
;------------------------------------------------------------------------
;   5 | return e;
;------------------------------------------------------------------------
           MVC     .S1     A8,RP             ; [A_S1] BARRIER
           LDD     .D1     *SP(16),A8        ; [A_D1]

           RET     .B1     ; [A_B]
||         ADDD    .D1     SP,0x8,SP         ; [A_D1]

           ; RETURN OCCURS {RP}              ; []
```

3.12 About the Application Binary Interface

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

The C7000 compiler supports only the Embedded Application Binary Interface (EABI) ABI, which works only with object files that use the ELF object file format and the DWARF debug format.

Copyright © 2026 Texas Instruments Incorporated

3.13 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 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 5.8.27 for information about the NO_HOOKS pragma.

Copyright © 2026 Texas Instruments Incorporated

Chapter 4

Optimizing Your Code

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

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

4.1 Invoking Optimization 56
4.2 Controlling Code Size Versus Speed 57
4.3 Performing File-Level Optimization (--opt_level=3 option) 57
4.4 Program-Level Optimization (--program_level_compile and --opt_level=3 options) 58
4.5 Automatic Inline Expansion (--auto_inline Option) 60
4.6 Link-Time Optimization (--opt_level=4 Option) 61
4.7 Optimizing Software Pipelining 62
4.8 Redundant Loops 67
4.9 Indicating Whether Certain Aliasing Techniques Are Used 68
4.10 Prevent Reordering of Associative Floating-Point Operations 68
4.11 Using Performance Advice to Optimize Code 69
4.12 Using the Interlist Feature With Optimization 70
4.13 Debugging and Profiling Optimized Code 70
4.14 What Kind of Optimization Is Being Performed? 71
4.15 Streaming Engine and Streaming Address Generator 75
4.16 Nested Loop Controller (NLC) 86

Copyright © 2026 Texas Instruments Incorporated

4.1 Invoking Optimization

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

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

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

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

- **--opt_level=off** or **-Ooff** (default if --opt_level option not used and --vectypes=off)
  - Performs no optimization
- **--opt_level=0** or **-O0** (default if --opt_level option not used and --vectypes=on)
  - Performs control-flow-graph simplification (Section 4.14.3)
  - Allocates variables to registers (Section 4.14.15)
  - Performs loop rotation (Section 4.14.10)
  - Eliminates unused code
  - Simplifies expressions and statements (Section 4.14.5)
  - Expands calls to functions declared as inline (Section 4.14.6)
- **--opt_level=1** or **-O1** Performs all --opt_level=0 (-O0) optimizations, plus:
  - Performs local copy/constant propagation (Section 4.14.4)
  - Removes unused assignments (Section 4.14.4)
  - Eliminates local common expressions (Section 4.14.4)
- **--opt_level=2** or **-O2** (default if --opt_level option used with no setting) Performs all --opt_level=1 (-O1) optimizations, plus:
  - Performs software pipelining (Section 4.7)
  - Performs loop optimizations (Section 4.14.9, Section 4.14.11, and Section 4.14.13)
  - Eliminates global common subexpressions (Section 4.14.4)
  - Eliminates global unused assignments (Section 4.14.4)
  - Converts array references in loops to incremented pointer form (Section 4.14.8)
  - Performs loop unrolling (Section 5.8.34)
- **--opt_level=3** or **-O3** Performs all --opt_level=2 (-O2) optimizations, plus:
  - Removes all functions that are never called (Section 4.4)
  - Simplifies functions with return values that are never used (Section 4.4)
  - Inlines calls to small functions (Section 3.10 and Section 4.5)
  - Reorders function declarations; the called functions' attributes are known when the caller is optimized
  - Propagates arguments into function bodies when all calls pass the same value in the same argument position
  - Identifies file-level variable characteristics (Section 4.4)
  - Performs other optimizations (Section 4.3 and Section 4.4)
- **--opt_level=4** or **-O4**
  - Performs link-time optimization (Section 4.6)

For details about how the --opt_level and --opt_for_speed options and various pragmas affect inlining, see Section 3.10.

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

Copyright © 2026 Texas Instruments Incorporated

Note

Do Not Lower the Optimization Level to Control Code Size: To reduce code size, do not lower the --opt_level optimization level. Instead, use the --opt_for_speed option to control the code size/ performance tradeoff. Higher optimization levels (--opt_level or -O) combined with low --opt_for_speed levels (0, 1 or 2) result in smaller code sizes.

4.2 Controlling Code Size Versus Speed

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

Table 4-1. Optimization Levels

Level	Description
--opt_for_speed=0 (-mf0)	Optimizes code size with a *high* risk of worsening or impacting performance.
--opt_for_speed=1 (-mf1)	Optimizes code size with a *medium* risk of worsening or impacting performance.
--opt_for_speed=2 (-mf2)	Optimizes code size with a *low* risk of worsening or impacting performance.
--opt_for_speed=3 (-mf3)	Optimizes code performance/speed with a *low* risk of worsening or impacting code size.
--opt_for_speed=4 (-mf4)	Optimizes code performance/speed with a *medium* risk of worsening or impacting code size. (default)
--opt_for_speed=5 (-mf5)	Optimizes code performance/speed with a *high* risk of worsening or impacting code size.

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

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

The --opt_level=3 option (aliased as the -O3 option) instructs the compiler to perform file-level optimization. You can use the --opt_level=3 option alone to perform general file-level optimization, or you can combine it with other options to perform more specific optimizations. The options listed in Table 4-2 work with --opt_level=3 to perform the indicated optimization:

Table 4-2. Options That You Can Use With --opt_level=3

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

Note

Do Not Lower the Optimization Level to Control Code Size: When trying to reduce code size, do not lower the level of optimization, as you might see an increase in code size. Instead, set the --opt_for_speed option to 0, 1, or 2 to reduce the code size.

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

When you invoke the compiler with the --opt_level=3 option , you can use the --gen_opt_info option to create an optimization information file that you can read. The number following the option denotes the level (0, 1, or 2). The resulting file has an .nfo extension. Use Table 4-3 to select the appropriate level to append to the option.

Table 4-3. 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_info=1 or --gen_opt_info=2 option in a command file or an environment variable. The --gen_opt_info=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 © 2026 Texas Instruments Incorporated

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

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

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

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

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

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

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

Note

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

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

Copyright © 2026 Texas Instruments Incorporated

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

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

Table 4-4. 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 4-5 lists the combinations of --call_assumptions levels and conditions that cause the compiler to revert to other --call_assumptions levels.

Table 4-5. Special Considerations When Using the --call_assumptions Option

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

In some situations when you use --program_level_compile and --opt_level=3, you *must* use a --call_assumptions option or the FUNC_EXT_CALLED pragma.

Copyright © 2026 Texas Instruments Incorporated

4.5 Automatic Inline Expansion (--auto_inline Option)

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

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

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

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

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

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

For information about interactions between command-line options, pragmas, and keywords that affect inlining, see Section 3.10.

Note

Some Functions Cannot Be Inlined: For a call-site to be considered for inlining, it must be legal to inline the function and inlining must not be disabled in some way. See the inlining restrictions in Section 3.10.2.

Note

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

Note

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

Copyright © 2026 Texas Instruments Incorporated

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

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

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

If you use --opt_level=4 (-O4), the --program_level_compile option cannot also be used, because link-time optimization provides the same optimization opportunities as program level optimization (Section 4.4). Link-time optimization provides the following benefits:

- Each source file can be compiled separately. One issue with program-level compilation is that it requires all source files to be passed to the compiler at one time. This often requires significant modification of a customer's build process. With link-time optimization, all files can be compiled separately.
- 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 some exceptions (see below). 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.

When using link-time optimization (--opt_level=4) for the C7000, link only source and object files that were compiled with the same --silicon_version and --mma_version options. Linking source and/or object files compiled with different --silicon_version or --mma_version options may cause the link to fail.

4.6.1 Option Handling

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

Some options are incompatible when performing link-time optimization. These are usually that which conflict on the command line as well, but can also be options that cannot be handled during link-time optimization. Notably, when using link-time optimization with the C7000 compiler, use only source and/or object files that were compiled with the same --silicon_version and --mma_version options.

4.6.2 Incompatible Types

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

Copyright © 2026 Texas Instruments Incorporated

4.7 Optimizing Software Pipelining

Software pipelining schedules instructions from a loop so that multiple iterations of the loop execute in parallel. At optimization levels --opt_level=2 (or -O2) and --opt_level=3 (or -O3), the compiler usually attempts to software pipeline your loops. The --opt_for_speed option also affects the compiler's decision to attempt to software pipeline loops. In general, code size and performance are better when you use the --opt_level=2 or --opt_level=3 options. (See Section 4.1.)

Figure 4-1 illustrates a software-pipelined loop. The stages of the loop are represented by A, B, C, D, and E. In this figure, a maximum of five iterations of the loop can execute at one time. The shaded area represents the loop *kernel*. In the loop kernel, all five stages execute in parallel. The area above the kernel is known as the *pipelined loop prolog*, and the area below the kernel is known as the *pipelined loop epilog*.

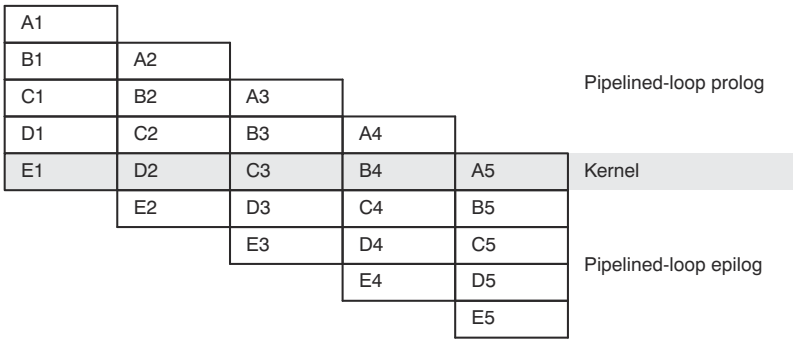


Figure 4-1. Software-Pipelined Loop

4.7.1 Turn Off Software Pipelining (--disable_software_pipeline Option)

At optimization levels --opt_level=2 (or -O2) and -O3, the compiler attempts to software pipeline your loops. You might not want your loops to be software pipelined for debugging reasons. Software-pipelined loops are sometimes difficult to debug because the code is not presented serially.

4.7.2 Software Pipelining Information

The compiler embeds software pipelined loop information in the .asm file. This information is used to optimize C/C++ code .

The software pipelining information appears as a comment in the .asm file before a loop and for the assembly optimizer the information is displayed as the tool is running. Example 4-1 illustrates the information that is generated for each loop.

The --debug_software_pipeline option adds additional information about the characteristics of the software pipeline loop, including the register usage at each cycle of the loop kernel. Information added by this option is likely useful only to very experienced C7000 users.

The following example shows some of the information that is emitted by default, that is, when the --debug_software_pipeline option is not used. When the --debug_software_pipeline option is not used, only the most useful and actionable information is displayed.

Copyright © 2026 Texas Instruments Incorporated

Example 4-1. Software Pipelining Information

```
;*------------------------------------------------------------------------------*
;*  SOFTWARE PIPELINE INFORMATION
;*    Loop found in file               : dts_biquad_c.c
;*    Loop source line                 : 123
;*    Loop opening brace source line   : 124
;*    Loop closing brace source line   : 142
;*
;*    Summary:
;*      - The compiler successfully software pipelined this loop.
;*      - The compiler cannot start scheduling attempts at an ii lower
;*        than 21 due to a loop-carried dependence.
;*
;*    Known Minimum Iteration Count    : 1
;*    Known Max Iteration Count Factor : 1
;*    Loop Carried Dependency Bound    : 21
;*    Partitioned Resource Bound       : 4 (pre-sched)
;*
;*    Searching for software pipeline schedule at ...
;*      ii = 21 Schedule found with 2 iterations in parallel
;*
;*    Partitioned Resource Bound(*)    : 5 (post-sched)
;*
;*    Resource Partition (may include "post-sched" split/spill moves):
;*
;*    Each bound value denotes the resource bound for instructions
;*    that can use one or more of the specified units.
;*
;*                                       A-side   B-side
;*
;*      Bound(.B)                          1        0
;*      Bound(.P)                          -        0
;*      Bound(.C)                          -        0
;*      Bound(.L)                          0        0
;*      Bound(.S)                          0        0
;*      Bound(.M)                          0        0
;*      Bound(.N)                          0        4
;*      Bound(.D)                          3        -
;*      Bound(.L/.S)                       3        1
;*      Bound(.M/.N)                       0        2
;*      Bound(.M/.S)                       0        0
;*      Bound(.L/.N)                       0        2
;*      Bound(.C/.S)                       0        0
;*      Bound(.C/.M)                       0        0
;*      Bound(.C/.L/.S)                    3        4
;*      Bound(.L/.M/.S)                    2        1
;*      Bound(.C/.D/.M)                    2        0
;*      Bound(.C/.L/.M/.S)                 2        3
;*      Bound(.L/.M/.N/.S)                 2        2
;*      Bound(.L/.S/.C/.M/.D)              4        4
;*
;*    Done
;*
;*    Minimum safe iteration count     : 1
```

4.7.2.1 Software Pipelining Information Terms

The terms defined below appear in the software pipelining information. Note that many of these items appear only if the --debug_software_pipeline option is used or only if that particular condition occurs.

- **Loop unroll factor.** The number of times the loop was unrolled specifically to increase performance.
- **Known minimum iteration count.** The minimum number of times the loop will be executed.
- **Known maximum iteration count.** The maximum number of times the loop will be executed.
- **Known max iteration count factor.** Factor that would always evenly divide the loop's iteration count. This information can be used to possibly unroll the loop.
- **Loop carried dependency bound.** The distance of the largest loop carry path. A loop carry path occurs when one iteration of a loop writes a value that must be read in a future iteration. Instructions that are part of the loop carry bound are marked with the ^ symbol.
- **Initiation interval** (ii). The number of cycles between the initiation of successive iterations of the loop. The smaller the initiation interval, the fewer cycles it takes to execute a loop.

Copyright © 2026 Texas Instruments Incorporated

- **Resource bound.** The minimum initiation interval, in cycles, imposed by the execution resources required for a loop iteration. The resource with the greatest demand relative to its availability determines this bound. For example, if four instructions require a .D unit and two .D units can execute instructions in parallel, the resource bound is two cycles.
- **Unpartitioned resource bound.** The best possible resource bound values before the instructions in the loop are partitioned to a particular side.
- **Partitioned resource bound (*).** The resource bound values after the instructions are partitioned.
- **Resource partition.** This table summarizes how instructions are partitioned. Each table entry has values for the A-side and B-side functional units. An asterisk marks entries that determine the resource bound value. For example, an entry of `.C/.L/.S units 0 9*` in the .asm file comment block indicates that there are 9 instructions that must be scheduled specifically among .C, .L, and .S units, with 0 on the A side and 9 on the B side.
- **Functional unit resource bounds.** Each `Bound(...)` entry specifies the minimum initiation interval imposed by the demand for a functional unit or group of functional units. Names in parentheses identify the functional units and combinations of units considered in calculating the bound. The bound accounts for the combined resource requirements of all instructions in the specified functional unitclasses and their subclasses. For example, `Bound(.L/.S/.C/.M/.D)` includes the .L/.S./C./M./D, .C/.L/.M/.S, .C/.L/.S, .L/.S, .C/.S, .C/.M, .M/.S, .C, .D, .L, .M, and .S subclasses. In the .asm file comment block, an entry of `Bound(.C/.L/.S) 3 4` shows that there is a resource bound of 3 on the A side and 4 on the B side imposed by instructions that may schedule on specifically .L, .S, .C, .C/.S, .L/.S, or .C/.L/.S.
- **Max amt of load speculation.** The maximum number of bytes that loads in this loop will speculate beyond the end of an array (behind or ahead). No user action is required here as the compiler will use load instructions that are safe to speculate.

4.7.2.2 Loop Disqualified for Software Pipelining Messages

The following messages appear if the loop is completely disqualified for software pipelining:

- **Bad loop structure**. This error is very rare and can stem from the following:
  - An asm statement inserted in the C code inner loop
  - Complex control flow such as GOTO statements and breaks
- **Loop contains a call**. Sometimes the compiler may not be able to inline a function call that is in a loop. Because the compiler could not inline the function call, the loop could not be software pipelined.
- **Too many instructions**. There are too many instructions in the loop to software pipeline.
- **Software pipelining disabled**. Software pipelining has been disabled by a command-line option, such as when using the --disable_software_pipeline option, not using the --opt_level=2 (or -O2) or --opt_level=3 (or -O3) option, or using the --opt_for_speed=0 or --opt_for_speed=1 option.
- **Uninitialized iteration counter**. The iteration counter may not have been set to an initial value.
- **Suppressed to prevent code expansion**. Software pipelining may be suppressed because of the --opt_for_speed=2 option. When the --opt_for_speed=2 option is used, software pipelining is disabled in less promising cases to reduce code size. To enable pipelining, use --opt_for_speed=4 or --opt_for_speed=5.
- **Cannot identify iteration counter**. The loop iteration counter could not be identified or was used incorrectly in the loop body.

Copyright © 2026 Texas Instruments Incorporated

4.7.2.3 Pipeline Failure Messages

The following messages can appear when the compiler or assembly optimizer is processing a software pipeline and it fails:

- **Address increment is too large**. An address register's offset must be adjusted because the offset is out of range of the offset addressing mode. You must minimize address register offsets.
- **Cannot allocate machine registers**. A software pipeline schedule was found, but it cannot allocate machine registers for the schedule. Simplification of the loop may help.

  The register usage for the schedule found at the given ii is displayed.

  - **Regs Live Always.** The number of values that must be assigned a register for the duration of the whole loop body. This means that these values must always be allocated registers for any given schedule found for the loop.
  - **Max Regs Live.** Maximum number of values live at any given cycle in the loop that must be allocated to a register. This indicates the maximum number of registers required by the schedule found.
  - **Max Cond Regs Live.** Maximum number of registers live at any given cycle in the loop kernel that must be allocated to a condition register.
- **Cycle count too high. Never profitable**. With the schedule that the compiler found for the loop, it is more efficient to use a non-software-pipelined version.
- **Did not find schedule**. The compiler was unable to find a schedule for the software pipeline at the given ii (iteration interval). You should simplify the loop and/or eliminate loop carried dependencies.
- **Iterations in parallel > minimum or maximum iteration count**. A software pipeline schedule was found, but the schedule has more iterations in parallel than the minimum or maximum loop iteration count. You must enable redundant loops or communicate the iteration information.
- **Register is live too long**. A register must have a value that exists (is live) for more than ii cycles. The compiler tries to insert move instructions to split register lifetimes that are longer than ii cycles , but may not always be successful.
- **Too many predicates live on one side**. The C7000 has predicate, or conditional, registers available for use with conditional instructions. There are six predicate registers. There are three on the A side and three on the B side. Sometimes the particular partition and schedule combination requires more than these available registers.
- **Schedule found with N iterations in parallel**. (This is not a failure message.) A software pipeline schedule was found with N iterations executing in parallel.
- **Iteration variable used in loop - Cannot adjust iteration count**. The loop iteration counter has a use in the loop other than as a loop iteration counter.
- **Unsafe schedule for irregular loop**. "Irregular" loops are non-downcounting loops with a known number of iterations, such as a while loop. Irregular loops may require transformations that execute instructions more times than called for by the loop. This error means the compiler was unable to find a schedule with instructions that are safe to over-execute, are guarded with a predicate, or have their effects undone after the loop. Try to rewrite the loop as a down-counting loop.

4.7.2.4 Register Usage Table Generated by the --debug_software_pipeline Option

The --debug_software_pipeline option places additional software pipeline feedback in the generated assembly file.

If software pipelining succeeds for a given loop, and the --debug_software_pipeline option was used during the compilation process, a register usage table is added to the software pipelining information comment block in the generated assembly code.

The numbers on each row represent the cycle number within the loop kernel.

Each column represents one register on the C7000. The registers are labeled in the first three rows of the register usage table and should be read columnwise.

An * in a table entry indicates that the register indicated by the column header is live on the kernel execute packet indicated by the cycle number labeling each row.

Copyright © 2026 Texas Instruments Incorporated

An example of the register usage table follows:

```
;*      Register Usage Tables:
;*
;*      +---------------------------------+---------------------------------+
;*      |              ASIDE              |              BSIDE              |
;*      +-----------------+--------+------+-----------------+--------+------+
;*      |      "Axx"      |  "ALx" | "AMx"|      "VBxx"     | "VBLx" | "VBMx" |
;*      +-----------------|--------|------|-----------------|--------|--------+
;*      |0000000000111111|00000000|00000000|0000000000111111|00000000|00000000|
;*      |0123456789012345|01234567|01234567|0123456789012345|01234567|01234567|
;*      +-----------------|--------|------|-----------------|--------|--------+
;*   0: |*  ***           |*       |***   |      *          |***     |******* |
;*   1: |*  ***           |**      | **   |     **          |***     |* ***** |
;*   2: |* ****           |*       |***   |   * **          |***     |*  **** |
;*   3: |******           |*       | **   |  ** **          |**      |*  **** |
;*   4: |** ***           |*       | **   |******           |***     |   **** |
;*   5: |** ***           |*       | **   |*** **           |**      |** **** |
;*   6: |*  ***           |*       | **   |*** **           |*       |******* |
;*   7: |*  ***           |*       |***   |*   **           |**      |******* |
;*      +-----------------+--------+------+-----------------+--------+--------+
;*
;*      +-----------------+                +--------+       +-------+
;*      |      "Dxx"      |                |  "Px"  |       |"CUCRx"|
;*      +-----------------+                +--------+       +-------+
;*      |0000000000111111|                |00000000|       |0000   |
;*      |0123456789012345|                |01234567|       |0123   |
;*      +-----------------+                +--------+       +-------+
;*   0: |*                |                |        |       |       |
;*   1: |*                |                |        |       |       |
;*   2: |*                |                |        |       |       |
;*   3: |*                |                |        |       |       |
;*   4: |*                |                |        |       |       |
;*   5: |*                |                |        |       |       |
;*   6: |*                |                |        |       |       |
;*   7: |*                |                |        |       |       |
;*      +-----------------+                +--------+       +-------+
```

This example shows that on cycle 0 (the first execute packet) of the loop kernel, registers A0, A3, A4, A5, A6, AL0, AM0, AM1, AM2, VB5, VBL0, VBL1, VBL2, VBM0, VBM1, VBM2, VBM3, VBM4, VBM5, VBM6, and D0 are all live during this cycle.

4.7.3 Collapsing Prologs and Epilogs for Improved Performance and Code Size

When a loop is software pipelined, a prolog and epilog are generally required. The prolog is used to pipe up the loop and epilog is used to pipe down the loop.

In general, a loop must execute a minimum number of iterations before the software-pipelined version can be safely executed. If the minimum known iteration count is too small, either a redundant loop is added or software pipelining is disabled. Collapsing the prolog and epilog of a loop can reduce the minimum iteration count necessary to safely execute the pipelined loop.

Collapsing can also substantially reduce code size. Some of this code size growth is due to the redundant loop. The remainder is due to the prolog and epilog.

The prolog and epilog of a software-pipelined loop consists of up to *p*-1 stages of length *ii*, where *p* is the number of iterations that are executed in parallel during the steady state and *ii* is the cycle time for the pipelined loop body. During prolog and epilog collapsing the compiler tries to collapse as many stages as possible. However, over-collapsing can have a negative performance impact. Thus, by default, the compiler attempts to collapse as many stages as possible without sacrificing performance. When the --opt_for_speed=2 or --opt_for_speed=1 options are invoked, the compiler increasingly favors code size over performance.

Copyright © 2026 Texas Instruments Incorporated

4.7.3.1 Speculative Execution

Collapsing of the prolog and epilog can usually reduce the minimum safe iteration count. If the minimum known iteration count is less than the minimum safe iteration count, a redundant loop is required. Otherwise, pipelining must be suppressed. Both of these values can be found in the comment block preceding a software pipelined loop when the --debug_software_pipeline option is used.

```
;*Known Minimum Iteration Count: 1
....
;*Minimum safe iteration count: 7
```

4.8 Redundant Loops

A loop iterates some number of times before the loop terminates. The number of iterations is called the *iteration count*. The variable that counts iterations is the *iteration counter*. When the iteration counter reaches a limit equal to the iteration count, the loop terminates. The Code Generation Tools use the iteration count to determine whether or not a loop can be pipelined. The structure of a software pipelined loop requires the execution of a minimum number of loop iterations (a minimum iteration count) in order to fill or prime the pipeline.

The minimum iteration count for a software pipelined loop is set by the number of iterations executing in parallel. In Figure 4-1, the minimum iteration count is five. In the following example A, B, and C are instructions in a software pipeline, so the minimum iteration count for this single-cycle software pipelined loop is three.

```
A
B     A
C     B     A     ←Three iterations in parallel = minimum iteration count
      C     B
            C
```

When the Code Generation Tools cannot determine the iteration count for a loop, then by default two loops and control logic are generated. The first loop is not pipelined, and it executes if the run-time iteration count is less than the loop's minimum safe iteration count. The second loop is the software pipelined loop, and it executes when the run-time iteration count is greater than or equal to the minimum iteration count. At any given time, one of the loops is a *redundant loop*. For example:

```
foo(N)  /* N is the iteration count */
{
   for (I=0; I < N; I++) /* I is the iteration counter */
}
```

After finding a software pipeline for the loop, the compiler transforms foo() as below, assuming the minimum iteration count for the loop is 3. Two versions of the loop would be generated and the following comparison would be used to determine which version should be executed:

```
foo(N)
{
   if (N < 3)
   {
      for (I=0; I < N; I++)   /* Unpipelined version */
   }
   else
   }
      for (I=0; I < N; I++)   /* Pipelined version   */
   }
}
foo(50); /* Execute software pipelined loop          */
foo(2);  /* Execute loop (unpipelined)*/
```

You may be able to help the compiler avoid producing redundant loops with the use of --program_level_compile --opt_level=3 (see Section 4.4) or the use of the MUST_ITERATE pragma (see Section 5.8.23).

Copyright © 2026 Texas Instruments Incorporated

4.9 Indicating Whether Certain Aliasing Techniques Are Used

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

The following sections describe some aliasing techniques that may be used in your code. These techniques are valid according to the ISO C standard and are accepted by the C7000 compiler; however, they prevent the optimizer from fully optimizing your code.

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

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

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

```
int *glob_ptr;
g()
{
    int  x = 1;
    int *p = f(&x);
    *p        = 5;    /* p aliases x    */
    *glob_ptr = 10;   /* glob_ptr aliases x */
    h(x);
}
int *f(int *arg)
{
    glob_ptr = arg;
    return arg;
}
```

4.10 Prevent Reordering of Associative Floating-Point Operations

The compiler freely reorders associative floating-point operations. If you do not wish to have the compiler reorder associative floating point operations, use the --fp_not_associative option. Specifying the --fp_not_associative option may decrease performance.

Copyright © 2026 Texas Instruments Incorporated

4.11 Using Performance Advice to Optimize Code

In some cases, the compiler can perform better optimization if the code aids the compiler by providing additional information in the code. In such cases, the compiler can prompt you to take certain actions to improve performance, by emitting "advice".

By default, the compiler emits diagnostic messages that suggest changes you can make to the source code to allow the compiler to perform more effective optimizations.

To control performance advice generation, use the following options:

--advice:performance	Instruct the compiler to emit advice to stdout (on by default). Compile with **--advice:performance=none** to fully disable generation of performance advice.
--advice:performance_file	Instruct the compiler to send any advice emitted to a file.
--advice:performance_dir	Instruct the compiler to send any advice emitted to a file in a specific directory.
--diag_suppress	Disable advice on a per-diagnostic basis. This option can also be used in the FUNCTION_OPTIONS pragma (see Section 5.8.20).

Example 1: The following command line sends output advice to a file named `myfile.adv`:

```
cl7x --advice:performance_file=myfile.adv loop.c
```

Example 2: The following command lines show two ways to send output advice to a file named `myfile.adv` in the `mydir` subdirectory:

```
cl7x --advice:performance_file=myfile.adv --advice:performance_dir=mydir loop.c
```

```
cl7x --advice:performance_file=mydir/myfile.adv loop.c
```

The subsections that follow describe the supported advice diagnostics.

4.11.1 Advice #35000: Use restrict to improve loop performance

```
advice #35000-D: (Performance) Consider adding the restrict qualifier to the declaration of
"variable" to improve loop performance.
```

This advice identifies opportunities for qualifying function parameters with `restrict` if doing so is likely to improve loop performance. Make sure that restrict-qualified pointers do not violate any of the limitations on such pointers; no other pointer can be used to access the object to which a restricted pointer points.

To see more information on using restrict, refer to Section 5.5.4

Copyright © 2026 Texas Instruments Incorporated

4.12 Using the Interlist Feature With Optimization

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

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

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

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

Note

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

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

4.13 Debugging and Profiling Optimized Code

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

The default optimization level is off.

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

If you are having trouble debugging loops in your code, you can use the --disable_software_pipeline option to turn off software pipelining. See Section 4.7.1 for more information.

4.13.1 Profiling Optimized Code

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

Copyright © 2026 Texas Instruments Incorporated

4.14 What Kind of Optimization Is Being Performed?

The C7000 C/C++ compiler uses a variety of optimization techniques to improve the execution speed of your C/C++ programs and to reduce their size. The following are some of the optimizations performed by the compiler:

Optimization	See
Cost-based register allocation	Section 4.14.1
Alias disambiguation	Section 4.14.2
Branch optimizations and control-flow simplification	Section 4.14.3
Data flow optimizations • Copy propagation • Common subexpression elimination • Redundant assignment elimination	Section 4.14.4
Expression simplification	Section 4.14.5
Inline expansion of functions	Section 4.14.6
Function symbol aliasing	Section 4.14.7
Induction variables and strength reduction	Section 4.14.8
Loop-invariant code motion	Section 4.14.9
Loop rotation	Section 4.14.10
Loop collapsing and loop coalescing	Section 4.14.11
Unroll-and-jam	Section 4.14.12
Vectorization	Section 4.14.13
Instruction scheduling	Section 4.14.14
Register variables	Section 4.14.15
Register tracking/targeting	Section 4.14.16
Software pipelining	Section 4.14.17

4.14.1 Cost-Based Register Allocation

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

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

This type of optimization is always enabled. It has no effect on the debuggability of the resulting application.

4.14.2 Alias Disambiguation

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

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

Copyright © 2026 Texas Instruments Incorporated

4.14.3 Branch Optimizations and Control-Flow Simplification

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

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

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

4.14.4 Data Flow Optimizations

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

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

4.14.5 Expression Simplification

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

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

4.14.6 Inline Expansion of Functions

The compiler replaces calls to small functions with inline code, saving the overhead associated with a function call as well as providing increased opportunities to apply other optimizations. For information about interactions between command-line options, pragmas, and keywords that affect inlining, see Section 3.10.

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

Copyright © 2026 Texas Instruments Incorporated

4.14.7 Function Symbol Aliasing

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

For example, consider the following:

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

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

For information about using the GCC function attribute syntax to declare function aliases, see Section 5.13.2

4.14.8 Induction Variables and Strength Reduction

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

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

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

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

4.14.9 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.

This type of optimization is enabled by the --opt_level=2 and higher optimization settings. This optimization can improve performance, but results in increased code size and reduced debuggability.

4.14.10 Loop Rotation

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

This type of optimization is enabled by the --opt_level=0 and higher optimization settings. This optimization can improve performance, but results in increased code size and reduced debuggability.

4.14.11 Loop Collapsing and Loop Coalescing

Under certain circumstances, the compiler combines a loop nest (for example, a "for" loop inside a "for" loop) when the compiler thinks this will lead to faster performance.

These types of optimization are enabled by the --opt_level=2 and higher optimization settings. These optimizations can improve performance, but result in increased code size and reduced debuggability.

Copyright © 2026 Texas Instruments Incorporated

4.14.12 Unroll-and-jam

The compiler can unroll an outer loop that encloses an innermost loop. This transformation makes an extra iteration of the outer loop, and as a result, there is another copy of the inner loop. The second "inner loop" is then "fused" back into the original inner loop. As a result, the fused inner loop performs two iterations of the outer loop for each execution of the inner loop. This transformation is called "unroll-and-jam" and can increase available parallelism and function unit utilization.

The compiler can perform unroll-and-jam if the compiler detects that there is not sufficient parallelism available in the inner loop to effectively utilize the computational resources on the CPU.

This type of optimization is performed if *both* the --opt_for_speed (-mf) option is set to level 3 or higher (level 4 is the default) and the --opt_level (-o) option is set to any level other than "off" (off is the default if --vectypes=off). This optimization can improve performance, but results in increased code size and reduced debuggability.

4.14.13 Vectorization (SIMD)

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

This type of optimization is performed if *both* the --opt_for_speed (-mf) option is set to level 2 or higher (level 4 is the default) and the --opt_level (-o) option is set to level 2 or higher. This optimization can improve performance, but results in increased code size and reduced debuggability.

4.14.14 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.

4.14.15 Register Variables

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

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

4.14.16 Register Tracking/Targeting

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

4.14.17 Software Pipelining

Software pipelining is a technique use to schedule from a loop so that multiple iterations of a loop execute in parallel. See Section 4.7 for more information.

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

Copyright © 2026 Texas Instruments Incorporated

4.15 Streaming Engine and Streaming Address Generator

The C7000 CPU has two unique features—the Streaming Engine (SE) and the Streaming Address Generator (SA)—which assist with bringing data to the CPU and computing addressing patterns.

4.15.1 Streaming Engine Overview

The Streaming Engines (SE) provide high-speed connections between the CPU functional units and the L2 controller. They can read from L2/L3/DDR, but are most efficient when used to read from L2.

A SE is a read-only data connection, which forwards and formats data on the fly. The SE presents data at up-to vector-width-bits per CPU clock in a holding-register. (See Table 5-7 for target-specific vector widths.) This holding register can be used like any other global or local register. Since the SE is a read-only engine, the Streaming Engine register can be used as input to instructions, but an instruction cannot write to the register.

Each SE has one vector-width holding register. Current variants of C7000 have two instances of the SE—named SE0 and SE1. Thus, there are two Streaming Engine registers that can be used as an input to instructions.

4.15.2 Theory of Operation for Streaming Engines and Streaming Address Generators

During typical operation, both the Streaming Engine (SE) and the Streaming Address Generator (SA) are conceptually modeled as a seven-level loop nest that calculates an offset. From the following model, some key points can be observed:

- An ICNT value is the number of iterations in a processing level of the SE or SA.
- A DIM value is the offset applied between each iteration of a level of processing, expressed as a number of elements. These values are often used to express the length of dimensions of data.
- After processing the next inner level, an outer level resets the offset and adds the DIM value.
- The elements of a returned vector must be adjacent during normal operation. Furthermore, for the innermost level, corresponding to ICNT0, processing is in a linear order. Elements are returned in vector-sized chunks of VECLEN elements. After each vector is processed in the innermost level, the SE or SA advances VECLEN elements. As such, the innermost two loops in the above model are together the innermost level when programming an SE or SA.

```
current_element_offset = 0;

/* ICNT5 programmed in the SE or SA template. */
for (i5 = 0; i5 < ICNT5; i5++)
{
    level5_offset = current_element_offset;
    /* ICNT4 programmed in the SE or SA template. */
    for (i4 = 0; i4 < ICNT4; i4++)
    {
        level4_offset = current_element_offset;
        /* ICNT3 programmed in the SE or SA template. */
        for (i3 = 0; i3 < ICNT3; i3++)
        {
            level3_offset = current_element_offset;
            /* ICNT2 programmed in the SE or SA template. */
            for (i2 = 0; i2 < ICNT2; i2++)
            {
                level2_offset = current_element_offset;
                /* ICNT1 programmed in the SE or SA template. */
                for (i1 = 0; i1 < ICNT1; i1++)
                {
                    level1_offset = current_element_offset;
                    /* ICNT0 programmed in the SE or SA template. */
                    /* VECLEN programmed in the SE or SA template. */
                    for (i0 = 0; i0 < ICNT0; i0 += VECLEN)
                    {
                        level0_offset = current_element_offset;
                        /* This inner loop demonstrates element processing of a vector.
                         * It is not considered a "level" of an SE or SA.
                         */
                        for (elem = 0; elem < VECLEN; elem++)
                        {
                            /* Element processing. For the SE, this builds a
```


Copyright © 2026 Texas Instruments Incorporated

```
                            * result vector element-by-element at the specified
                            * offset from the base address, including
                            * performing vector predication. For the SA, this
                            * builds a predicate.
                            */
                           process_element(current_element_offset);

                           /* DIM0 cannot be programmed. It is always 1. */
                           DIM0 = 1;
                           current_element_offset += DIM0;
                       }
                       /* Vector processing. For the SE, this yields the vector
                        * that was built in the SE register (e.g. SE0). For the
                        * SA, this yields the offset that the vector starts at
                        * in the SA register (e.g. SA0) and a vector predicate
                        * in the SA predicate register (e.g. PSA0).
                        */
                      process_return_vector_data(level0_offset);
                   }
                   /* DIM1 programmed in the SE or SA template. */
                   current_element_offset = level1_offset + DIM1;
               }
               /* DIM2 programmed in the SE or SA template. */
               current_element_offset = level2_offset + DIM2;
           }
           /* DIM3 programmed in the SE or SA template. */
           current_element_offset = level3_offset + DIM3;
       }
       /* DIM4 programmed in the SE or SA template. */
       current_element_offset = level4_offset + DIM4;
   }
   /* DIM5 programmed in the SE or SA template. */
   current_element_offset = level5_offset + DIM5;
}
```

4.15.3 Streaming Address Generator Overview

The Streaming Address Generator (SA) is a multi-dimensional offset computation engine. Computing multi-dimensional offsets using deeply nested loops costs CPU cycles. Current C7000 variants offer up to four address generators—named SA0, SA1, SA2, and SA3—to compute offsets up to six dimensions. The SA can be used in conjunction with load/store instructions.

Programming an SA is similar to programming an SE. The difference is that an SA provides only address offsets, whereas an SE provides the actual data.

4.15.4 Advantages of Using Streaming Engines and Streaming Address Generators

There are several advantages to using one or more Streaming Engines and/or Streaming Address Generators:

- Both hardware features allow you to specify the iteration count (ICNT) and offset (DIM) for each dimension of data, up to six dimensions.
- Both hardware features compute address offsets automatically. There is no need for the compiler to generate address calculation instructions, which typically reduce loop performance. Using one or more Streaming Engines can allow the compiler to software pipeline more code, leading to improved performance.
- Streaming Engines prefetch data from above the L2 cache into memory that is close to the CPU.
- Streaming Engines provide several additional data formatting and pattern access features.

Copyright © 2026 Texas Instruments Incorporated

4.15.5 Interface for Streaming Engine and Streaming Address Generator

The Streaming Engine (SE) and Streaming Address Generator (SA) intrinsic interface is defined in `c7x_strm.h`, which is included as part of the C7000 runtime support and is included automatically by `c7x.h`.

As described below, basic utilization of the SE and SA relies on parameter templates that must be pre-configured before the SE or SA can be opened and used. The templates of the SE and the SA both have iteration count (ICNT), offset (DIM), and vector length (VECLEN) values. These values are used to determine offsets as a stream is read and advanced.

For the SE, the element size is also configured directly into the setup flags and is then used in the SE address calculation. However, the SA does not work this way. For the SA, the offset is scaled based on the element size of the load or store instruction that uses the SA. This allows the same SA to be effectively used by different load or store instructions that have different element sizes.

4.15.6 Parameter Template Configuration

A parameter template is a variable of type `__SE_TEMPLATE_v1` or `__SA_TEMPLATE_v1` that is used to specify counters, sizes, and flags that control the behavior of the Streaming Engine (SE) and Streaming Address Generator (SA). The iteration counters (ICNT) and dimension offsets (DIM) for up to six levels of nesting apply to both the SE and SA. Functional flags are specific to either the SE or the SA. The parameter templates are effectively vectors of 64-bit values.

Note that `__SE_TEMPLATE_v1` and `__SA_TEMPLATE_v1` are defined as separate types.

To initialize parameters for the `__SE_TEMPLATE_v1` type, call `__gen_SE_TEMPLATE_v1()`.

```
__SE_TEMPLATE_v1 se_params = __gen_SE_TEMPLATE_v1();
```

To initialize parameters for the `__SA_TEMPLATE_v1` type, call `__gen_SA_TEMPLATE_v1()`.

```
__SA_TEMPLATE_v1 sa_params = __gen_SA_TEMPLATE_v1();
```

After initialization, configure the parameter template using assignments like the following:

- Set Iteration Counter for Level 0 (Dimension is always implied to be the element width)

  ```
  p.ICNT0 = <value>;
  ```

- Set Iteration Counter and Dimension Size for Level 1-5

  ```
  p.ICNT<n> = <value>;
  p.DIM<n> = <value>;
  ```

- Set DECDIM width

  ```
  p.DECDIM<n>.WIDTH = <value>;
  ```

- Set LEZR Count

  ```
  p.LEZR_CN = <value>;
  ```


Copyright © 2026 Texas Instruments Incorporated

For example, to configure a parameter vector for an SE with four dimensions:

```
// Setup Template Vector Based on Settings and Open the Stream
// Based on Iteration Counters and Dimensions (in Terms of # of Elems)
__SE_TEMPLATE_v1 params = __gen_SE_TEMPLATE_v1();
params.ICNT0 = 4;
params.ICNT1 = 2;
params.DIM1  = 4;
params.ICNT2 = 2;
params.DIM2  = 8;
params.ICNT3 = 4;
params.DIM3  = -16;
```

The counters and dimension sizes for unused dimensions do not need to be configured as long as the DIMFMT flag is set to the proper format flag. This flag tells the hardware to ignore uninitialized data in these unused fields.

For the Streaming Engine, the following additional flags can be configured within a variable defined with the __SE_TEMPLATE_v1 type. These flags are documented in the c7x_strm.h file.

```
ELETYPE
TRANSPOSE
PROMOTE
GRPDUP
VECLEN
ELDUP
DECIM
DIR
DIMFMT
DECDIM1
DECDIM2
LEZR
```

For example, the following statements configure SE parameters using constants provided in c7x_strm.h.

```
se_params.DIMFMT    = __SE_DIMFMT_4D;
se_params.DIR       = __SE_DIR_INC;
se_params.TRANSPOSE = __SE_TRANSPOSE_OFF;
se_params.DECIM     = __SE_DECIM_OFF;
se_params.VECLEN    = __SE_VECLEN_4ELEMS; // 4 ELEMENTS
se_params.ELETYPE   = __SE_ELETYPE_32BIT;
se_params.PROMOTE   = __SE_PROMOTE_OFF;
se_params.GRPDUP    = __SE_GRPDUP_OFF;
se_params.ELDUP     = __SE_ELEDUP_OFF;
se_params.DECDIM1   = __SE_DECDIM0;
se_params.DECDIM2   = __SE_DECDIM0;
se_params.LEZR      = __SE_LEZR_OFF;
```

For the Streaming Address Generator, the following additional flags can be configured within a variable defined with the __SA_TEMPLATE_v1 type.

```
VECLEN
DIMFMT
DECDIM1
DECDIM1SD
DECDIM2
DECDIM2SD
```

For example, the following statements configure SA parameters using constants provided in c7x_strm.h.

```
sa_params.VECLEN    = __SA_VECLEN_4ELEMS; // 4 ELEMENTS
sa_params.DIMFMT    = __SA_DIMFMT_4D;
sa_params.DECDIM1   = __SA_DECDIM0;
sa_params.DECDIM2   = __SA_DECDIM0;
```


Copyright © 2026 Texas Instruments Incorporated

4.15.7 Using the Streaming Engine

Once the Streaming Engine has been configured, a corresponding stream can be opened and used. The following API is provided to facilitate this, which take the parameter template as well as the starting memory address from which the hardware should fetch the stream data:

- Open Streaming Engine 0: `__SE0_OPEN(void *addr, __SE_TEMPLATE_v1 param);`
- Open Streaming Engine 1: `__SE1_OPEN(void *addr, __SE_TEMPLATE_v1 param);`

The Streaming Engines are accessed using the following API, which will return a vector of data according to the given type, which is used to cast the type of the data being fetched by the stream:

- Read Streaming Engine 0 (and advance): `__SE0(type), __SE0ADV(type)`
- Read Streaming Engine 1 (and advance): `__SE1(type), __SE1ADV(type)`

Note

The type passed to the Streaming Engine access APIs must be specified exactly as <*base_type*> for scalars and <*base_type*><*length*> for vectors. The <*base_type*> must be one of "char", "uchar", "cchar", "short", "ushort", "cshort", "int", "uint", "cint", "float", "cfloat", "long", "ulong", or "double". For example, __SE0(int16). The specified type cannot be larger than the size of a vector register. Typedef types are not allowed. For additional type support and template support in C++ when reading from a Streaming Engine, see "strm_eng" in Section 5.16.

Streaming Engines are closed using the corresponding close API:

- Close Streaming Engine 0: `__SE0_CLOSE();`
- Close Streaming Engine 1: `__SE1_CLOSE();`

Example for a stream with four dimensions:

```
// OPEN STREAMING ENGINE 1 AT startaddr WITH PARAMETER TEMPLATE params
__SE1_OPEN((void*)startaddr, params);
// READ THE STREAM AND ADVANCE THE COUNTERS
for (I0 = 0; I0 < 8; I0++)
{
    uint8 Vout;
    Vout.lo = __SE1ADV(uint4);
    Vout.hi = __SE1ADV(uint4);
    Vresult += Vout;
}
// CLOSE THE STREAM
__SE1_CLOSE();
```

Note

Streaming from the provided memory address implies a contract in which the program promises never to modify the data in any area of memory that can be reached by that Streaming Engine during its lifetime.

Copyright © 2026 Texas Instruments Incorporated

4.15.7.1 Hard-Coded Intrinsic Operands with the Streaming Engine

A small set of C7000 instructions, specifically some VFIR and VMATMPY instructions, accept *only* Streaming Engine operands because they are hard-wired to the SE. This includes accesses of a single SE (e.g. SE0) as well as accesses of a pair of SE streams (e.g. SE1:SE0). These instructions have a special intrinsic interface, which is defined using an enumeration called `SE_REG` or `SE_REG_PAIR` that must be used with the intrinsic. Because of this interface, these instructions can only be accessed using direct-mapped, low-level intrinsics.

- SE_REG

```
// Use the following for SE_REG operands.
enum SE_REG
{
    SE_REG_0     = 0, // READ SE0
    SE_REG_0_ADV = 1, // READ SE0 AND ADVANCE
    SE_REG_1     = 2, // READ SE1
    SE_REG_1_ADV = 3  // READ SE1 AND ADVANCE
};
```

- SE_REG_PAIR

```
// Use the following for SE_REG_PAIR operands.
enum SE_REG_PAIR
{
    SE_REG_PAIR_0     = 0, // READ SE0 AND SE1
    SE_REG_PAIR_0_ADV = 1  // READ SE0 AND SE1 AND ADVANCE BOTH STREAMS
};
```

4.15.8 Using the Streaming Address Generator

Similar to the SE, once the Streaming Address Generator (SA) has been configured, a corresponding SA can be opened and used. The following API is provided to facilitate this, which takes the parameter template as input:

- Open Streaming Address Generator 0 (SA0): `__SA0_OPEN(__SA_TEMPLATE_v1 param);`
- Open Streaming Address Generator 1 (SA1): `__SA1_OPEN(__SA_TEMPLATE_v1 param);`
- Open Streaming Address Generator 2 (SA2): `__SA2_OPEN(__SA_TEMPLATE_v1 param);`
- Open Streaming Address Generator 3 (SA3): `__SA3_OPEN(__SA_TEMPLATE_v1 param);`

The Streaming Address Generators are accessed using the following API. Note that for Streaming Address Generators, the base address is given as an input to the read operation, and the return value is a pointer to the location that is then used by a memory load or memory store operation. This is because, unlike the SE, an SA simply provides an offset added to the given base address. The given type is used to cast the type of the data loaded or stored through the SA.

- Read SA0 (and advance): `__SA0(type, baseptr)`, `__SA0ADV(type, baseptr)`
- Read SA1 (and advance): `__SA1(type, baseptr)`, `__SA1ADV(type, baseptr)`
- Read SA2 (and advance): `__SA2(type, baseptr)`, `__SA2ADV(type, baseptr)`
- Read SA3 (and advance): `__SA3(type, baseptr)`, `__SA3ADV(type, baseptr)`

Note

The type specified to the Streaming Address Generator access APIs cannot be larger than the size of a vector register. Additionally, for non-complex vectors, the element type cannot be larger than 64 bits. For complex scalars and complex vectors, the complex component type cannot be larger than 64 bits. (For example, the complex component type of cshort is short.) Typedef types and qualified types are allowed. For example, the following are all legal:

```
__SA0(int2, baseptr)
__SA0(__int2, baseptr)
__SA0(const __int2, baseptr)
```

For additional template support in C++ when reading from a Streaming Address Generator, see "strm_agen" in Section 5.16.

Copyright © 2026 Texas Instruments Incorporated

Streaming Address Generators are closed using the corresponding close API:

- Close SA0: `__SA0_CLOSE();`
- Close SA1: `__SA1_CLOSE();`
- Close SA2: `__SA2_CLOSE();`
- Close SA3: `__SA3_CLOSE();`

When the accessor APIs shown above are used by themselves, the compiler will match a basic load or store operation for them, depending on whether they are on the left-hand-side (LHS) or right-hand-side (RHS) of an assignment operator.

The following is an example of an SA-returned pointer dereferenced by itself on both the LHS and RHS to copy data from one location to another. Because the SA returns a pointer, dereferencing it on the RHS generates a load, and dereferencing it on the LHS generates a store.

```
// OPEN THE STREAMS, SA0 AND SA1
__SA0_OPEN(params);
__SA1_OPEN(params);
// COPY DATA FROM *(src_addr+SA1) TO *(dst_addr+SA0)
for (I0 = 0; I0 < 8; I0++)
{
    // COMPILER MATCHES VECTOR LOAD AND STORE FOR FOUR WORD
DATA
    *__SA0ADV(uint8, dst_addr) = *__SA1ADV(uint8, src_addr);
}
// CLOSE THE STREAMS
__SA0_CLOSE();
__SA1_CLOSE();
```

However, the SA accessor APIs can also be given as input to a load or store intrinsic, as in the following example. In this case, since the SA simply returns a pointer that is not dereferenced, the compiler will not attempt to match a corresponding basic vector load or store and will instead use the load or store indicated by the intrinsic.

```
// OPEN THE STREAMS, SA0 AND SA1
__SA0_OPEN(params);
__SA1_OPEN(params);
// COPY DATA (WITH UNPACK + PACK) FROM *(src_addr+SA1) TO *(dst_addr+SA0)
for (I0 = 0; I0 < 8; I0++)
{
   ulong8 data = __vload_unpack_long(__SA1ADV(uint8, src_addr));
   __vstore_packl(__SA0ADV(uint8, dst_addr), data);
}
// CLOSE THE STREAMS
__SA0_CLOSE();
__SA1_CLOSE();
```

Note

If the Streaming Address Generator is used with a base pointer that is flagged as `restrict`, then derivations based on that pointer (base + offset), as calculated by the SA, are also assumed to be `restrict`. This means that SA-generated pointers, and all loads and stores that use them, are assumed to not alias other memory pointers.

Therefore, using the SA based on `restrict` base pointers implies a contract in which the program promises never to modify the data in any area of memory that can be reached by that Streaming Address Generator during its lifetime.

Copyright © 2026 Texas Instruments Incorporated

4.15.8.1 Vector Predication for Streaming Address Generators

Each Streaming Address Generator (SA) has an implicit predicate that is set when storing to memory based on the number of valid vector elements. However, expressing this behavior using C semantics is not possible, because the compiler does not see or track how or when the hardware does the predication, since the predication is based upon the SA configuration and may vary for each invocation of the store.

Note

SAs on all C7000 ISAs may be configured to allow predicated SA stores. Predicated SA loads are not supported by the C7100 ISA. However, for C7120 and later ISAs, SAs may be configured to allow predicated SA loads.

In order to express this functionality in C, the compiler provides an explicit semantic framework that you can leverage. The framework explicitly predicates a store operation with a predicate value extracted from the corresponding SA. If the compiler detects that the extracted predicate type matches the SA accessor (and corresponding memory operation), the compiler then attempts to collapse this down in order to leverage the SA's implicit predication capability.

The API to extract the predicate value into a value of type `__vpred` includes the following:

- Extract VPRED from SA0 and scale according to specified type: `__SA0_VPRED(type)`
- Extract VPRED from SA1 and scale according to specified type: `__SA1_VPRED(type)`
- Extract VPRED from SA2 and scale according to specified type: `__SA2_VPRED(type)`
- Extract VPRED from SA3 and scale according to specified type: `__SA3_VPRED(type)`

The following is an example of this use:

```
__SA0_OPEN(params);
for (I = 0; I < ((ROW * COL) / SIMD); I++)
{
    vpred sa0_vp = __SA0_VPRED(int16);             // EXTRACT SA0 PREDICATE
    int16 *addr  = __SA0ADV(int16, (int16 *)data); // EXTRACT SA0 ADDRESS
    __vstore_pred(sa0_vp, addr, data);             // USE PREDICATED INTRINSIC
}
__SA0_CLOSE();
```

Note

As indicated in the example, extraction of the vector predicate associated with a particular Streaming Address Generator must be done outside of the call to the store intrinsic. If this were not the case, according to C rules, the order in which parameters of a call are evaluated would be unspecified. Therefore doing this:

```
__vstore_pred(__SA0_VPRED(int16), __SA0ADV(int16, (int16 *)data);
```

results in undefined behavior because SA0ADV modifies the value returned by SA0_VPRED().

Note

Regardless of the type used in the predicate extraction API or the SA access, an SA advance will always increment based on its preconfigured state. The given typename does not impact how an SA advance is done.

Copyright © 2026 Texas Instruments Incorporated

4.15.8.2 Predicated vs. Unpredicated Streaming Address Stores and Loads

The Streaming Address Generators (SA) may be configured to cause the generation of SA vector predicates containing 0s that may be used to mask part of a load or store. When the SA is configured this way and a load or store is based on the SA, it is unspecified whether vector predication will be applied to the operation unless vector predication is explicitly specified in the program source code.

The following definitions apply to the code examples for SA and memory operations in this section:

- *Predicated:* Part of an operation *may be* masked out, as determined by the SA configuration.
- *Unpredicated:* Part of an operation *will not be* masked out, as determined by the SA configuration.

Given these definitions, predicated operations are actually a super-set of unpredicated operations, because all unpredicated operations can be represented as predicated operations.

A pointer P1 is defined as *based on* another pointer P2 if the value of P1 is derived from P2. For example, if `ptr = __SA0()`, then `ptr` is based on SA0. Similarly, if `ptr1 = __SA0()` and `ptr2 = (ptr1 + 0)`, both `ptr1` and `ptr2` are based on SA0.

The following C examples show various store and load operations. For each operation type, examples that produce well-defined or unspecified behavior are shown for SA configurations that either require unpredicated operations or allow predicated operations.

Store operations

- SA is configured to require unpredicated stores. (Parts of an operation will *not* be masked out.)
  - Well-defined examples:
    - ```
      *__SA0ADV(int16, baseptr) = data;              // Normal store
      ```
    - ```
      __vstore_packl(__SA0ADV(int8, baseptr), data); // Specialized store
      ```
    - ```
      int16 *ptr = __SA0ADV(int16, baseptr);
      *ptr = data;
      ```
  - Examples that result in unspecified behavior:

    None, because the SA is configured for unpredicated operations only. The predicated store operations shown below are also well-defined with this configuration, although they may be slower.
- SA is configured to allow predicated stores. (Parts of an operation may be masked out.)
  - Well-defined examples:
    - ```
      __vpred vp = __SA0_VPRED(int16);
      int16 *ptr = __SA0ADV(int16, baseptr);
      __vstore_pred(vp, ptr, data);           // Normal store with explicit predication
      ```
    - ```
      __vpred vp = __SA0_VPRED(int8);
      int8 *ptr = __SA0ADV(int8, baseptr);
      __vstore_pred_packl(vp, ptr, data);     // Specialized store with explicit predication
      ```
  - Examples that invoke unspecified behavior:
    - ```
      *__SA0ADV(int16, baseptr) = data;              // May be predicated
      ```
    - ```
      __vstore_packl(__SA0ADV(int8, baseptr), data); // May be predicated
      ```
    - ```
      int16 *ptr = __SA0ADV(int16, baseptr);
      *ptr = data;                                   // May be predicated
      ```


Copyright © 2026 Texas Instruments Incorporated

Load operations

- SA is configured to require unpredicated loads. (Parts of an operation will *not* be masked out.)
  - Well-defined examples:
    - ```
      int16 x = *__SA0ADV(int16, baseptr);                          // Normal load
      ```
    - ```
      ushort32 x = __vload_unpack_short(__SA0ADV(uchar32, baseptr); // Specialized load
      ```
    - ```
      int16 *ptr = __SA0ADV(int16, baseptr);
      int16 x    = *ptr;
      ```
  - Examples that invoke unspecified behavior:

    None, because the SA is configured for unpredicated operations. The predicated load operations shown below are also well-defined with this configuration, although they may be slower.
- SA is configured to allow predicated loads. (Parts of an operation may be masked out.)
  - Well-defined examples:
    - ```
      __vpred vp = __SA0_VPRED(int16);
      int16 x = *__SA0ADV(int16, baseptr);                          // Normal load
      x = select(vp, x, (int16)(0));                                // Apply predication
      ```
    - ```
      __vpred vp = __SA0_VPRED(int16);
      int16 *ptr = __SA0ADV(int16, baseptr);
      int16 x    = __vload_pred(vp, ptr);              // Normal load, explicit predication
      ```
    - ```
      __vpred vp = __SA0_VPRED(uchar32);
      uchar32 *ptr = __SA0ADV(uchar32, baseptr);
      ushort32 x = __vload_pred_unpack_short(vp, ptr); // Specialized load, explicit predication
      ```
  - Examples that invoke unspecified behavior:
    - ```
      int16 x = *__SA0ADV(int16, baseptr);                          // May be predicated
      ```
    - ```
      ushort32 x = __vload_unpack_short(__SA0ADV(uchar32, baseptr); // May be predicated
      ```
    - ```
      int16 *ptr = __SA0ADV(int16, baseptr);
      int16 x = *ptr // May be predicated
      ```

4.15.9 Automatic Use of Streaming Engine and Streaming Address Generator (--auto_stream Option)

The compiler can be directed to automatically use the Streaming Engines (SE) and/or the Streaming Address Generators (SA) using the --auto_stream option. This behavior can be controlled with the following settings for the --auto_stream option:

- **--auto_stream=off:** Disables automatic use of the SE and SA. (Default for C7100 and C7120)
- **--auto_stream=saving:** Enables automatic use of the SE and SA with context saving. Use this option if an SE or SA can be open when a function call is made. Using this option is safe, but the result can be slightly slower than with --auto_stream=no_saving and can have increased stack size. (Default for C7504 and later devices)
- **--auto_stream=no_saving:** Enables automatic use of the SE and SA without context saving. Use this option if an SE or SA is never open when a function call is made. Using this option is less safe than with --auto_stream=saving, but the result can be slightly faster and can have reduced stack size.

C7100 and C7120 devices support only "off" and "no_saving" modes. Enable optimization manually using the --auto_stream=no_saving option. These devices provide no support for SE or SA context switching, so --auto_stream=off by default.

Newer devices, such as C7504, support all three modes and --auto_stream=saving is the default.

Copyright © 2026 Texas Instruments Incorporated
      ```

With --auto_stream enabled, memory accesses are converted in nested loops that have addressing patterns that can fit into an SE or SA configuration template. For example, suppose you have the following code:

```
void example1(char *in, char *restrict out, int len1, int len2)
{
    for (int i = 0; i < len1; i++)
        for (int j = 0; j < len2; j++)
            out[i*len1 + j] = in[i*len1 + j];
}
```

With --auto_stream enabled, this code is transformed to be equivalent to the following SE configuration on C7504 devices after being vectorized:

```
__SE_TEMPLATE_v1 tmplt = __gen_SE_TEMPLATE_v1();
tmplt.ICNT0 = 32;
tmplt.ICNT1 = (len2>>5)+((len2&0x1f) != 0);
tmplt.DIM1 = 32;
tmplt.ICNT2 = len1;
tmplt.DIM2 = len1;
tmplt.VECLEN = __SE_VECLEN_32ELEMS;
tmplt.DIMFMT = __SE_DIMFMT_3D;
```

For further information about how a stream is programmed, see Section 4.15.6.

4.15.9.1 Correctness of Automatic Use of SE and SA

Unlike the example in the previous section, the following example is not transformed, because `len1` and `len2` can potentially exceed the size of the 32-bit fields of the SE and the loop counters can exceed 32-bit values:

```
void example2(char *in, char *restrict out, long len1, long len2)
{
    for (long i = 0; i < len1; i++)
        for (long j = 0; j < len2; j++)
            out[i*len1 + j] = in[i*len1 + j];
}
```

In practice, such addressing patterns almost always map to a stream, although edge cases are still possible. Such cases include, but are not limited to:

- ICNT values exceeding the unsigned 32-bit range.
- DIM values exceeding the signed 32-bit range.
- Additions or multiplies in addressing exceeding the signed 32-bit range.
- Addressing exceeding the range of INT_MIN to INT_MAX elements.

For further information about how a stream is programmed, see Section 4.15.6.

The --assume_addresses_ok_for_stream option allows the compiler to ignore edge cases such as these. Using this option allows `example2` to be transformed in the same way as `example1`.

Incorrect code can be generated if the --auto_stream=no_saving option is used when an SE or SA is open and a function call is made. In this case, the state of the SE or SA that is open is lost if that SE or SA is used automatically by the compiler.

Incorrect code can also be generated if --auto_stream is used and L1D is configured and used as SRAM. In this case, attempting to use the SE to access L1D fails.

4.15.9.2 Tuning Automatic Use of SE and SA

Automatic use of the SE and SA occurs only if the compiler believes that transforming within a loop or loop nest is profitable, which is primarily related to loop iteration counts. As such, using the PROB_ITERATE and MUST_ITERATE pragmas helps to guide this transformation. For further information about these pragmas, see Section 5.8.23 and Section 5.8.30.

Additionally, the compiler does not use an SE or SA if an SE or SA is already used in a function.

Copyright © 2026 Texas Instruments Incorporated

To control the behavior of automatic SE and SA on a function-by-function basis, use the FUNCTION_OPTIONS pragma. The following C++ example controls automatic SE and SA behavior for the function that follows the pragma on C7100 devices. For C code, add the function name as the first argument to the pragma. For further information about the FUNCTION_OPTIONS pragma, see Section 5.8.20.

```
#pragma FUNCTION_OPTIONS("--auto_stream=no_saving --assume_addresses_ok_for_stream")
```

4.16 Nested Loop Controller (NLC)

In order to better facilitate nested loop *coalescing*, in which an outer loop level containing instructions is merged into an inner loop level, the C7000 architecture provides a feature known as the Nested Loop Controller (NLC). This feature implements hardware loop control for no more than one level of loop nesting. This allows the compiler to coalesce loop levels while predicating the execution of outer loop instructions in the inner loop. This reduces loop control overhead and provides for better function unit resource utilization for software pipelined loops. Loop coalescing is something that the compiler attempts to do automatically if the operation is both legal and profitable. Profitability is determined by a heuristic.

There are two pragmas that allow you to explicitly enable or disable coalescing of specific nested loops. These pragmas can only be applied to loops and must appear immediately before a loop construct in C/C++:

```
#pragma COALESCE_LOOP
#pragma NO_COALESCE_LOOP
```

4.16.1 Obstacles That May Inhibit Use of NLC

As stated above, the compiler must check for legality before attempting to coalesce the loop. If the compiler cannot guarantee that an inner loop is executed at least one time, then it will not coalesce the loop, even if you have disabled the profitability heuristic.

To inform the compiler that a loop will always be executed, use a `MUST_ITERATE` pragma just prior to the loop body:

```
#pragma MUST_ITERATE(1,65535,)
```

This pragma tells the compiler that the loop executes at least once and no more than 65,535 times.

Copyright © 2026 Texas Instruments Incorporated

Chapter 5

C/C++ Language Implementation

The C language supported by the C7000 was developed by a committee of the American National Standards Institute (ANSI) and subsequently adopted by the International Standards Organization (ISO).

The C++ language supported by the C7000 is defined by the ANSI/ISO/IEC 14882:2014 standard with certain exceptions.

5.1 Characteristics of C7000 C 88
5.2 Characteristics of C7000 C++ 92
5.3 Data Types 93
5.4 File Encodings and Character Sets 98
5.5 Keywords 98
5.6 C++ Exception Handling 104
5.7 Register Variables and Parameters 105
5.8 Pragma Directives 106
5.9 The _Pragma Operator 126
5.10 Application Binary Interface 127
5.11 Object File Symbol Naming Conventions (Linknames) 127
5.12 Changing the ANSI/ISO C/C++ Language Mode 128
5.13 GNU and Clang Language Extensions 130
5.14 Operations and Functions for Vector Data Types 136
5.15 C7000 Intrinsics 148
5.16 C7000 Scalable Vector Programming 151

Copyright © 2026 Texas Instruments Incorporated

5.1 Characteristics of C7000 C

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

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

The C language is also described in the second edition of Kernighan and Ritchie's *The C Programming Language* (K&R). The compiler can also accept many of the language extensions found in the GNU C compiler (see Section 5.13).

The compiler supports some features of C99 in the default relaxed ANSI mode with C89 support. It supports all language features of C99 in C99 mode. See Section 5.12.

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

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

Unsupported features of the C library are:

- The run-time library has minimal support for wide characters. The type wchar_t is implemented as unsigned int (32 bits). The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h>, but does not include all the functions specified in the standard. See Section 5.4 for information about extended and multibyte character sets.
- The run-time library includes the header file <locale.h>, but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hard-coded to the behavior of the C locale, and attempting to install a different locale by way of a call to setlocale() will return NULL.
- Some run-time functions and features in the C99 specification are not supported. See Section 5.12.

5.1.1 Implementation-Defined Behavior

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

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

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

J.3.1 Translation

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

J.3.2 Environment

- The compiler does not support multibyte characters in identifiers, string literals, or character constants. There is no mapping from multibyte characters to the source character set. However, the compiler accepts multibyte characters in comments. See Section 5.4 for details (5.1.1.2)
- The name of the function called at program startup is "main". (5.1.2.1)

Copyright © 2026 Texas Instruments Incorporated

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

J.3.3. Identifiers

- The compiler does not support multibyte characters in identifiers. See Section 5.4 for details. (6.4.2)
- The number of significant initial characters in an identifier is unlimited. (5.2.4.1, 6.4.2)

J.3.4 Characters

- The number of bits in a byte (CHAR_BIT) is 8. See Section 5.3 for details about data types. (3.6)
- The execution character set is the same as the basic execution character set: plain ASCII. (5.2.1)
- The values produced for the standard alphabetic escape sequences are as follows. (5.2.2):

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

- The value of a char object into which any character other than a member of the basic execution character set has been stored is the ASCII value of that character. (6.2.5)
- Plain char is identical to signed char. (6.2.5, 6.3.1.1)
- The source character set and execution character set are both plain ASCII, so the mapping between them is one-to-one. The compiler accepts multibyte characters in comments. See Section 5.4 for details. (6.4.4.4, 5.1.1.2)
- The compiler currently supports only one locale, "C". (6.4.4.4)
- The compiler currently supports only one locale, "C". (6.4.5)

J.3.5 Integers

- No extended integer types are provided. (6.2.5)
- Negative values for signed integer types are represented as two's complement, and there are no trap representations. (6.2.6.2)
- No extended integer types are provided, so there is no change to the integer ranks. (6.3.1.1)

Copyright © 2026 Texas Instruments Incorporated

- When an integer is converted to a signed integer type which cannot represent the value, the value is truncated (without raising a signal) by discarding the bits which cannot be stored in the destination type; the lowest bits are not modified. (6.3.1.3)
- Right shift of a signed integer value performs an arithmetic (signed) shift. The bitwise operations other than right shift operate on the bits in exactly the same way as on an unsigned value. That is, after the usual arithmetic conversions, the bitwise operation is performed without regard to the format of the integer type, in particular the sign bit. (6.5)

J.3.6 Floating point

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

J.3.7 Arrays and pointers

- When converting a 64-bit pointer to a 32-bit integer, the pointer is truncated. When converting a pointer to a long, the pointer is considered an unsigned long of the same size, and the normal conversion rules apply.
- When converting a pointer to a long or vice versa, if the bitwise representation of the destination can hold all of the bits in the bitwise representation of the source, the bits are copied exactly. (6.3.2.3)
- The size of the result of subtracting two pointers to elements of the same array is the size of ptrdiff_t, which is defined in Section 5.3. (6.5.6)

J.3.8 Hints

- When the optimizer is used, the register storage-class specifier is ignored. When the optimizer is not used, the compiler will preferentially place register storage class objects into registers to the extent possible. The compiler reserves the right to place any register storage class object somewhere other than a register. (6.7.1)
- The inline function specifier is ignored unless the optimizer is used. For other restrictions on inlining, see Section 3.10.2. (6.7.4)

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

- A "plain" int bit-field is treated as a signed int bit-field. (6.7.2, 6.7.2.1)
- In addition to _Bool, signed int, and unsigned int, the compiler allows char, signed char, unsigned char, signed short, unsigned short, signed long, unsigned long, and enum types as bit-field types. (6.7.2.1)
- Bit-fields may not straddle a storage-unit boundary. (6.7.2.1)
- Bit-fields are allocated in endianness order within a unit. See Section 6.2.2. (6.7.2.1)
- Non-bit-field members of structures are aligned as specified in Section 6.2.1. (6.7.2.1)
- The integer type underlying each enumerated type is described in Section 5.3.1. (6.7.2.2)

Copyright © 2026 Texas Instruments Incorporated

J.3.10 Qualifiers

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

J.3.11 Preprocessing directives

- Include directives may have one of two forms, " " or < >. For both forms, the compiler will look for a real file on-disk by that name using the include file search path. See Section 3.5.2. (6.4.7)
- The value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (both are ASCII). (6.10.1)
- The compiler uses the file search path to search for an included < > delimited header file. See Section 3.5.2. (6.10.2)
- The compiler uses the file search path to search for an included " " delimited header file. See Section 3.5.2. (6.10.2)
- There is no arbitrary nesting limit for #include processing. (6.10.2)
- See Section 5.8 for a description of the recognized non-standard pragmas. (6.10.6)
- The date and time of translation are always available from the host. (6.10.8)

J.3.12 Library functions

- Almost all of the library functions required for a hosted implementation are provided by the TI library, with exceptions noted in Section 5.12.1. (5.1.2.1)
- The format of the diagnostic printed by the assert macro is "Assertion failed, (*assertion macro argument*), file *file*, line *line*". (7.2.1.1)
- No strings other than "C" and "" may be passed as the second argument to the setlocale function. (7.11.1.1)
- No signal handling is supported. (7.14.1.1)
- The +INF, -INF, +inf, -inf, NAN, and nan styles can be used to print an infinity or NaN. (7.19.6.1, 7.24.2.1)
- The output for %p conversion in the fprintf or fwprintf function is the same as %x of the appropriate size. (7.19.6.1, 7.24.2.1)
- The termination status returned to the host environment by the abort, exit, or _Exit function is not returned to the host environment. (7.20.4.1, 7.20.4.3, 7.20.4.4)
- The system function is not supported. (7.20.4.6)

J.3.13 Architecture

- The values or expressions assigned to the macros specified in the headers float.h, limits.h, and stdint.h are described along with the sizes and format of integer types are described in Section 5.3. (5.2.4.2, 7.18.2, 7.18.3)
- The number, order, and encoding of bytes in any object are described in Section 6.2.1. (6.2.6.1)
- The value of the result of the sizeof operator is the storage size for each type, in terms of bytes. See Section 6.2.1. (6.5.3.4)

Copyright © 2026 Texas Instruments Incorporated

5.2 Characteristics of C7000 C++

The C7000 compiler supports C++ as defined in the ANSI/ISO/IEC 14882:2014 standard (C++14), including these features:

- Complete C++ standard library support, with exceptions noted below.
- Templates
- Exceptions, which are enabled with the --exceptions option; see Section 5.6.
- Run-time type information (RTTI) cannot be disabled. The metadata for a virtual class refers to the class' type_info object so that the class' actual type can be determined at run-time.

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

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

Copyright © 2026 Texas Instruments Incorporated

5.3 Data Types

Table 5-1 lists the size, representation, and range of each scalar data type for the C7000 compiler. Many of the range values are available as standard macros in the header file limits.h.

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

Table 5-1. C7000 C/C++ Data Types

			Range	
Type	**Size**	**Representation**	**Minimum**	**Maximum**
char, signed char	8 bits	ASCII	-128	127
unsigned char	8 bits	ASCII	0	255
cchar(2)	16 bits	ASCII	-128 for real and imaginary portions separately	127 for real and imaginary portions separately
_Bool, bool	8 bits	ASCII	0 (false)	1 (true)
short	16 bits	Binary	-32 768	32 767
unsigned short	16 bits	Binary	0	65 535
cshort(2)	32 bits	Binary	-32 768 for real and imaginary portions separately	32 767 for real and imaginary portions separately
int, signed int	32 bits	Binary	-2 147 483 648	2 147 483 647
unsigned int, wchar_t	32 bits	Binary	0	4 294 967 295
cint(2)	64 bits	Binary	-2 147 483 648 for real and imaginary portions separately	2 147 483 647 for real and imaginary portions separately
long, signed long	64 bits	Binary	-9 223 372 036 854 775 808	9 223 372 036 854 775 808
unsigned long	64 bits	Binary	0	18 446 744 073 709 551 615
long long, signed long long	64 bits	Binary	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	Binary	0	18 446 744 073 709 551 615
clong(2)	128 bits	Binary	-9 223 372 036 854 775 808 for real and imaginary portions separately	9 223 372 036 854 775 807 for real and imaginary portions separately
enum (3)	varies	Binary	varies	varies
fp8_e4m3_	8 bits	Non-standard IEEE	0.001 953 125(1)	448
bf16_	16 bits	Non-standard IEEE	1.175 494e-38(1)	3.38 953 14e+38
float	32 bits	IEEE 32-bit	1.175 494e-38(1)	3.40 282 346e+38
cfloat, float complex (2)	64 bits	Array of 2 IEEE 32-bit	1.175 494e-38 for real and imaginary portions separately	3.40 282 346e+38 for real and imaginary portions separately
double	64 bits	IEEE 64-bit	2.22 507 385e-308(1)	1.79 769 313e+308
cdouble, double complex (2)	128 bits	Array of 2 IEEE 64-bit	2.22 507 385e-308 for real and imaginary portions separately	1.79 769 313e+308 for real and imaginary portions separately
long double	64 bits	IEEE 64-bit	2.22 507 385e-308(1)	1.79 769 313e+308
long double complex (2)	128 bits	Array of 2 IEEE 64-bit	2.22 507 385e-308 for real and imaginary portions separately	1.79 769 313e+308 for real and imaginary portions separately
pointers, references, pointer to data members	64 bits	Binary	0	0xFFFFFFFFFFFFFFFF

(1) For floating-point types, the minimum value is the smallest positive normal or subnormal (denormal) value, as applicable. Subnormal values can represent smaller nonzero numbers, but provide reduced precision. The minimum values shown for `bf16_`, `float` and `double` are minimum precision *normal* values. The minimum value shown for `fp8_e4m3_` is the *subnormal* value. The maximum value shown for floating-point types is the maximum finite value. See Table 5-2 for details about minimum floating-point values.

(2) Complex data types that use TI syntax are described in Section 5.3.2. Complex data types that use the C99 _Complex keyword are described in Section 5.5.1.

(3) For details about the size of an enum type, see Section 5.3.1.

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

Copyright © 2026 Texas Instruments Incorporated

Note

We recommend that your code use the C standard integer types `int64_t` and `int32_t` (et al.) when specific data type sizes are needed for portability across different devices and compilers. These standard integer types are defined in `stdint.h`, which is included as part of the C standard library support included the Runtime Support Library.

The C7000 adopts the LP64 representational convention. That is, the type `int` is 32 bits, while `long` and `pointer` types are 64 bits.

The C7000 compiler provides two built-in custom TI IEEE floating-point number types. The `fp8_e4m3_` (or `fp8_e4m3`) type has 1 sign bit, 4 bits of exponent, and 3 bits of mantissa. The `bf16_` (or `bf16`) type has 1 sign bit, 8 bits of exponent, and 7 bits of mantissa. These types are supported only by the assignment operator. Literals are not supported, but you can reinterpret to/from these types using the `__as_<type>()` syntax. Use intrinsics to perform arithmetic with and to numerically convert to/from these types.

The minimum positive floating point values allowed are as follows. Values are approximate in decimal notation; hexadecimal values are exact. Subnormal values may be processed normally or treated as zero values depending on the processor's operating mode.

Table 5-2. Minimum Positive Floating-Point Values

Type	Minimum Positive Normal	Minimum Positive Subnormal
fp8_e4m3_	0.015625 (`0x1p-6`)	0.001953125 (`0x1p-9`)
bf16_	1.17549435e-38 (`0x1p-126`)	9.18355e-41 (`0x1p-133`)
float	1.17549435e-38 (`0x1p-126`)	1.40130e-45 (`0x1p-149`)
double	2.2250738585072014e-308 (`0x1p-1022`)	4.9406564584124654e-324 (`0x1p-1074`)
long double	2.2250738585072014e-308 (`0x1p-1022`)	4.9406564584124654e-324 (`0x1p-1074`)

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

Table 5-3. C Language Standard Types

Type	Definition
size_t	unsigned long
ptrdiff_t	long
wchar_t	unsigned int
wint_t	int
va_list	char *

5.3.1 Size of Enum Types

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

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

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

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

Copyright © 2026 Texas Instruments Incorporated

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

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

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

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

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

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

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

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

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

```
enum sl { a, b, c, d = -1, e = UINT_MAX };
```

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

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

5.3.2 Vector, Complex, and Complex Vector Data Types

The C7000 C/C++ compiler supports the use of TI vector data types, complex data types, and complex vector data types in C/C++ source files.

Note

C99 syntax: This section describes the TI compiler syntax for vectors and complex numbers. If, instead, you want to use the _Complex keyword described by C99, see Section 5.5.1. Although both syntaxes store data in the same way, the functions and operations used to access them are generally incompatible. See the example at the end of Section 5.14.2.1 for how to access a TI syntax complex vector from an array of C99 complex values.

Vector Data Types: Vector data types are similar to an array, in that a vector contains a specified number of elements of a specified type. However, the vector length can only be one of 2, 3, 4, 8, 16, 32, 64, or 128. Wherever possible, intrinsics that act upon vectors are optimized to make use of efficient single instruction, multiple data (SIMD) instructions on the device.

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

Support for vector data types is enabled by default. You can use the --vectypes=off compiler option if you want to disable the vector data type names that are *not* prefixed with a double-underscore. For example, the `__int4`

Copyright © 2026 Texas Instruments Incorporated

type is always available, but the --vectypes=off option disables the `int4` type. Note that the tables and examples that follow use type names without the double-underscore prefix.

All of the vector data types and related built-in functions that are supported in the C7000 programming model are specified in the "c7x.h" header file.

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

The C7000 implementation of vector data types and operations is somewhat similar to the OpenCL C language specification. The C7000 programming model provides the following built-in vector data types:

Table 5-4. Vector Data Types

Type	Description	Maximum Elements
char*n*	A vector of *n* 8-bit signed integer values.	128
uchar*n*	A vector of *n* 8-bit unsigned integer values.	128
short*n*	A vector of *n* 16-bit signed integer values.	64
ushort*n*	A vector of *n* 16-bit unsigned integer values.	64
int*n*	A vector of *n* 32-bit signed integer values.	32
uint*n*	A vector of *n* 32-bit unsigned integer values.	32
long*n*	A vector of *n* 64-bit signed integer values.	16
ulong*n*	A vector of *n* 64-bit unsigned integer values.	16
float*n*	A vector of *n* 32-bit single-precision floating-point values.	32
double*n*	A vector of *n* 64-bit double-precision floating-point values.	16
fp8_e4m3_*n*	A vector of *n* 8-bit float values with 4 bits of exponent and 3 bits of mantissa.	128
bf16_*n*	A vector of *n* 16-bit float values with 8 bits of exponent and 7 bits of mantissa.	64
bool*n*	A vector of *n* 8-bit unsigned integer Boolean values.	128

where n can be a vector length of 2, 3, 4, 8, 16, 32, 64, or 128.

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

Vectors types are aligned on a boundary equal to the total size of the vector's elements up to 64 bits. Any vector type with a total size of more than 64 bits is aligned to a 64-bit boundary (8 bytes). For example, a short2 has a total size of 32 bits and is aligned on a 4-byte boundary. A long2 has a total size of 128 bits and is aligned on an 8-byte boundary.

Complex Data Types: The C/C++ compiler supports types that store complex values. Each complex type contains a real component and an imaginary component. The real part occupies the lower address in memory. Both components are of the same type. For example, a cfloat has a real component of type float and an imaginary component of type float. Complex types are aligned to a boundary equal to the alignment of its component types.

A prefix of 'c' is used to indicate complex type names.

There are two keywords for each complex type, one with a double-underscore prefix (for example, `__cint`) and one without (for example, `cint`). The keywords with a double-underscore prefix are always available, even in strict ANSI mode (--strict_ansi). The keywords without a double-underscore prefix are available by default, but can be disabled by using the --vectypes=off compiler option. The table that follows and examples in this user guide assume that --vectypes=on, which is the default.

The following complex data types are supported. Complex unsigned types are not supported.

Copyright © 2026 Texas Instruments Incorporated

Table 5-5. Complex Data Types

Type	Description
cchar	A single pair of 8-bit signed integer values.
cshort	A single pair of 16-bit signed integer values.
cint	A single pair of 32-bit signed integer values.
clong	A single pair of 64-bit signed integer values.
cfloat	A single pair of 32-bit floating-point values.
cdouble	A single pair of 64-bit floating-point values.

Be aware of the difference between type names such as `cchar` and `cchar2`. If the name of a complex type is followed by a number, it is a complex vector of the specified length.

Valid operations on complex types are: addition, subtraction, multiplication, division, negation, increment, decrement, comparison for equality. Arithmetic operations on complex types conform to the usual arithmetic rules of complex numbers. Increment and decrement operators affect only the real component. The following operations are not allowed on complex types: relational comparison (<, >), modulo (%), bitwise operations (& | ~), logical operations (&& ||).

Complex Vector Data Types: The C/C++ compiler also provides an extension for representing vectors of complex types. The complex vector types are as follows:

Table 5-6. Complex Vector Data Types

Type	Description	Maximum Elements
cchar*n*	A vector of *n* pairs of 8-bit signed integer values.	64
cshort*n*	A vector of *n* pairs of 16-bit signed integer values.	32
cint*n*	A vector of *n* pairs of 32-bit signed integer values.	16
clong*n*	A vector of *n* pairs of 64-bit signed integer values.	8
cfloat*n*	A vector of *n* pairs of 32-bit floating-point values.	16
cdouble*n*	A vector of *n* pairs of 64-bit floating-point values.	8

where n can be a vector length of 1, 2, 4, 8, 16, 32, or 64. Note that 128 is not a valid vector length for complex vector types. For example, a "cfloat2" is a vector of 2 complex floats. Its length is 2 and its size is 128 bits. Each "cfloat2" vector element contains a real float and an imaginary float.

Note that some C7000 targets do not support all of these complex vector types.

In addition, type names with a double-underscore prefix (for example, `__cchar2` and `__cint8`) are provided as aliases for the complex vector types.

Note

Vectors cannot be passed to variadic functions (stdarg.h) and cannot be passed to printf().

The vector width on a C7000 device varies depending on the target. The following table shows the widths of vectors in number of bytes on currently released C7000 devices.

Table 5-7. Vector Widths

Silicon Version	Vector Width in Bytes	Vector Predicate Width
C7100	64	8
C7120	64	8
C7504	32	4
C7524	32	4
C7604	128	16

Copyright © 2026 Texas Instruments Incorporated

Note

While the compiler supports all vector types on all C7000 devices, performance may suffer for operations on vector types larger than the target's vector registers.

For information about operators and built-in functions used with vector data types, see Section 5.14.

5.4 File Encodings and Character Sets

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

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

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

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

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

5.5 Keywords

The C7000 C/C++ compiler supports all of the keywords (such as const and volatile) specified in the corresponding language standard (C89, C99, C11, C++14), depending on whether the language mode is C or C++ and whether strict ANSI mode is enabled.

Some of the keywords that the compiler supports are:

- const
- inline
- register
- restrict
- volatile

Some keywords are not available in strict ANSI mode. For example, the inline and restrict keywords, which are keywords in C99 but not C89, are not available as keywords in C89 strict ANSI mode. These features are still available as extensions to the language in the form of __inline and __restrict, as are other keywords that are introduced in later versions of the standard. You can use the keyword names with leading underscores whether or not you are using strict ANSI mode. See Section 5.12.3 for a discussion of why the non-underscore versions of these keywords are not accepted in strict ANSI mode.

The TI compiler also adds its own keywords, such as "cregister". As with the standard keywords, this keyword is not available in strict ANSI mode, but the alternate form "__cregister" is always available. The same is true for other target-specific keywords when using strict ANSI mode; the alternate form prefixed by two underscores is always available.

Copyright © 2026 Texas Instruments Incorporated

5.5.1 The complex Keyword

Note

TI syntax: This section describes the C99 syntax for storing complex numbers. If, instead, you want to use the TI syntax for storing vectors and complex number, see Section 5.3.2. Although both syntaxes store data in the same way, the functions and operations used to access them are generally incompatible. See the example at the end of Section 5.14.2.1 for how to access a TI syntax complex vector from an array of C99 complex values.

To store complex data using the C99 syntax, use syntax like the following:

```
float _Complex var1; /* always allowed */
float complex var1;  /* allowed only if you include complex.h */
```

If you include the <complex.h> header file, complex support is available for all C/C++ modes, including relaxed and strict ANSI modes and C89 and C99. The <complex.h> header file implements math operations and functions for complex data types.

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

```
float complex x;
```

5.5.2 The const Keyword

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

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

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

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

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

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

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

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


Copyright © 2026 Texas Instruments Incorporated

5.5.3 The __cregister Keyword

The compiler extends the C/C++ language by adding the __cregister keyword to allow high level language access to control registers.

When you use the __cregister keyword on an object, the compiler compares the name of the object to a list of standard control registers (see Table 5-8). If the name matches, the compiler generates the code to reference the control register. If the name does not match, the compiler issues an error.

The following control registers are declared in the `c7x.h` header file. In addition, a large number of Extended Control Registers (ECRs) are declared in `c7x_ecr.h`.

Table 5-8. Control Registers for C7000

Category	Register	Description
General Control Registers	CPUID	CPU ID register
	PMR	Power management register
	DNUM	DSP core number register
	TSC	Time-stamp counter register
	TSR	Task state register
	RP	Return pointer register
	BPCR	Branch predictor control register
	STSC	Shadow time stamp counter register
Computation Control Registers	FPCR	Floating-point control register
	FSR	Flag status register
	GPLY	Galois polynomial register
	GFPGFR	Galois field polynomial generator function register
Event Control Registers	DEPR	Debug event priority register
	IESET	Internal exception event set register
	ESTP_SS	Event service table pointer register, secure supervisor
	ESTP_S	Event service table pointer register, supervisor
	ESTP_GS	Event service table pointer register, guest supervisor
	ECSP_SS	Event context save pointer register, secure supervisor
	ECSP_S	Event context save pointer register, supervisor
	ECSP_GS	Event context save pointer register, guest supervisor
	TCSP	Task context save pointer
	RXMR_SS	Returning execution mode register, secure supervisor
	RXMR_S	Returning execution mode register, supervisor
	AHPEE	Highest priority enabled event register, currently in service
	PHPEE	Highest priority enabled event register, pending
	IEBER	Internal event broadcast enable register
	IERR	Internal exception report register
	IEAR	Internal exception address register
	IESR	Internal exception status register
	IEDR	Internal exception data register
	TCR	Test count register
	TCCR	Test count config register
	GMER	Guest mode enable register
	UMER	User mask enable register
	SPBR	Stack pointer boundary register

Copyright © 2026 Texas Instruments Incorporated

Table 5-8. Control Registers for C7000 (continued)

Category	Register	Description
	UFCMR	User flag clear mask register
	IPE	Inter-processor event register
Lookup Table and Histogram Control Registers	LTBR0 to LTBR3	Lookup table base address registers
	LTCR0 to LTCR3	Lookup table configuration registers
	LTER	Lookup table enable register
Debug Control Registers	DBGCTXT	Debug context (overlay) register
	ILCNT	Inner loop counter register
	OLCNT	Outer loop counter initial value register
	LCNTFLG	16-bit predicate flags register
	SCRB	Scoreboard bits register

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

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

To use the control registers in Table 5-8, you must declare each register as follows. The c7x.h include file defines all the control registers through this syntax:

extern __cregister volatile unsigned int *register* ;

Once you have declared the register, you can use the register name directly.

Example 5-1. Define and Use Control Registers

```
extern __cregister volatile unsigned long __CPUID;
int main()
{
    printf("__CPUID = 0x%lx\n", __CPUID);
}
```


Copyright © 2026 Texas Instruments Incorporated

5.5.3.1 Evaluating Flags in the Flag Status Register (FSR) After Floating Point Operations

The Flag Status Register (FSR), which contains bits representing floating point status, can be accessed using the `__get_FSR(type)` API, which is defined in `c7x.h`. The API takes a *type* argument, which refers to a valid scalar or vector floating point type (except for "float3") that is being used with the floating point operation.

The API returns an "OR" of the data bits for all pertinent vector lanes. The result is an 8-bit value containing the following fields:

- Bit 7: SAT - NOT SUPPORTED FOR C7000 OPERATIONS
- Bit 6: UNORD - Compare result is unordered Floating Point Flag
- Bit 5: DEN - Source is a Denorm Floating Point Flag
- Bit 4: INEX - Result is inexact Floating Point Flag
- Bit 3: UNDER - Result is underflow Floating Point Flag
- Bit 2: OVER - Result is overflow Floating Point Flag
- Bit 1: DIV0 - Divide by zero Floating Point Flag
- Bit 0: INVAL - Invalid Operations Floating Point Flag

For example:

```
float4 a = ... ;
float4 b = ... ;
float4 c = a * b;
uint8_t fsr_val = __get_FSR(float4);
```

The `__get_FSR(type)` API is provided to make accessing the FSR easier. The register value is divided into 8-bit chunks. Each 8-bit chunk corresponds to a 64-bit vector slice of data in either the input or output data, depending on the operation being performed. A 64-bit slice may consist of a 64-bit double, two 32-bit float, four bf16_, or eight fp8_e4m3_ values that are OR'd together by the hardware.

However, for vector operations, while this "OR" is done for every 64-bit slice, the results for all 64-bit slices are not OR'd together by the hardware. The reason for this is that when partial vectors are used (with a size less than the native vector register width), the upper lanes of a vector are considered invalid and are ignored and therefore should not be reflected in the final FSR result. To ensure that only the information pertinent to the valid lanes of a vector are reflected, the API allows users to specify the scalar or vector type of the data they are working with. The API then ensures that only the valid 64-bit vector slices are OR'd together through a sequence of instructions to produce a final 8-bit result. All invalid lanes are therefore ignored.

Note

Using the `__get_FSR(type)` API results in performance degradations. This is because the API inserts a sequence of instructions to ensure that only the valid vector lanes are reflected in the final result. The API also prevents loop vectorization throughout a function in which it is used because vectorization would change the number of valid vector lanes in ways the user is not able to track.

Copyright © 2026 Texas Instruments Incorporated

5.5.4 The restrict Keyword

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

The "restrict" keyword is a C99 keyword, and cannot be accepted in strict ANSI C89 mode. Use the "__restrict" keyword if the strict ANSI C89 mode must be used. See Section 5.12.

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

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

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

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

5.5.5 The volatile Keyword

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

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

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

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

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


Copyright © 2026 Texas Instruments Incorporated

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

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

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

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

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

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

5.6 C++ Exception Handling

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

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

Exception handling requires support in the run-time-support library, which come in exception-enabled and exception-disabled forms; you must link with the correct form. When using automatic library selection (the default), the linker automatically selects the correct library Section 11.3.1.1. If you select the library manually, you must use run-time-support libraries whose name contains _eh if you enable exceptions.

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

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

Copyright © 2026 Texas Instruments Incorporated

5.7 Register Variables and Parameters

The C/C++ compiler treats register variables (variables defined with the register keyword) differently, depending on whether you use the --opt_level (-O) option.

- **Compiling with optimization**

  The compiler ignores any register definitions and allocates registers to variables and temporary values by using an algorithm that makes the most efficient use of registers.
- **Compiling without optimization**

  If you use the register keyword, you can suggest variables as candidates for allocation into registers. The compiler uses the same set of registers for allocating temporary expression results as it uses for allocating register variables.

The compiler attempts to honor all register definitions. If the compiler runs out of appropriate registers, it frees a register by moving its contents to memory. If you define too many objects as register variables, you limit the number of registers the compiler has for temporary expression results. This limit causes excessive movement of register contents to memory.

Any object with a scalar type (integral, floating point, or pointer) can be defined as a register variable. The register designator is ignored for objects of other types, such as arrays.

The register storage class is meaningful for parameters as well as local variables. Normally, in a function, some of the parameters are copied to a location on the stack where they are referenced during the function body. The compiler copies a register parameter to a register instead of the stack, which speeds access to the parameter within the function.

For more information about register conventions, see Section 6.3.

Copyright © 2026 Texas Instruments Incorporated

5.8 Pragma Directives

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

- CALLS (See Section 5.8.1)
- CLINK (See Section 5.8.2)
- CODE_ALIGN (See Section 5.8.4)
- CODE_SECTION (See Section 5.8.5)
- COALESCE_LOOP (See Section 5.8.3)
- DATA_ALIGN (See Section 5.8.6)
- DATA_MEM_BANK (See Section 5.8.7)
- DATA_SECTION (See Section 5.8.8)
- diag_suppress, diag_remark, diag_warning, diag_error, diag_default, diag_push, diag_pop (See Section 5.8.9)
- FORCEINLINE (See Section 5.8.10)
- FORCEINLINE_RECURSIVE (See Section 5.8.11)
- FUNC_ALWAYS_INLINE (See Section 5.8.12)
- FUNC_CANNOT_INLINE (See Section 5.8.13)
- FUNC_EXT_CALLED (See Section 5.8.14)
- FUNC_IS_PURE (See Section 5.8.15)
- FUNC_IS_SYSTEM (See Section 5.8.16)
- FUNC_NEVER_RETURNS (See Section 5.8.17)
- FUNC_NO_GLOBAL_ASG (See Section 5.8.18)
- FUNC_NO_IND_ASG (See Section 5.8.19)
- FUNCTION_OPTIONS (See Section 5.8.20)
- INTERRUPT (See Section 5.8.21)
- LOCATION (See Section 5.8.22)
- MUST_ITERATE (See Section 5.8.23)
- NOINIT (See Section 5.8.24)
- NOINLINE (See Section 5.8.25)
- NO_COALESCE_LOOP (See Section 5.8.26)
- NO_HOOKS (See Section 5.8.27)
- once (See Section 5.8.28)
- pack (See Section 5.8.29)
- PERSISTENT (See Section 5.8.24)
- PROB_ITERATE (See Section 5.8.30)
- RETAIN (See Section 5.8.31)
- SET_CODE_SECTION (See Section 5.8.32)
- SET_DATA_SECTION (See Section 5.8.32)
- STRUCT_ALIGN (See Section 5.8.33)
- UNROLL (See Section 5.8.34)
- WEAK (See Section 5.8.35)

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

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

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

Copyright © 2026 Texas Instruments Incorporated

5.8.1 The CALLS Pragma

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

The CALLS pragma is used by the compiler to embed debug information about indirect calls in object files. Using the CALLS pragma on functions that make indirect calls enables such indirect calls to be included in calculations for such functions' inclusive stack sizes. For more information on generating function stack usage information, see the -cg option of the Object File Display Utility in Section 13.1.

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

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

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

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

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

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

The GCC-style "calls" function attribute (see Section 5.13.2), which has the same effect as the CALLS pragma, has the following syntax:

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

5.8.2 The CLINK Pragma

Note

The CLINK pragma has no effect because conditional linking is enabled by default. It is documented here because the .clink directive is generated as a result of using this pragma. See Section 11.2.1 for more information about conditional linking.

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

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma CLINK

The RETAIN pragma has the opposite effect of the CLINK pragma. See Section 5.8.31 for more details.

5.8.3 The COALESCE_LOOP Pragma

In order to facilitate nested loop coalescing, in which an outer loop level containing instructions is merged into an inner loop level, the C7000 architecture provides a feature known as the Nested Loop Controller (NLC). This feature implements hardware loop control for no more than one level of loop nesting. This allows the compiler to coalesce loop levels while predicating the execution of outer loop instructions in the inner loop. This reduces

Copyright © 2026 Texas Instruments Incorporated

loop control overhead and provides for better function unit resource utilization for software pipelined loops. Loop coalescing is something that the compiler attempts to do automatically if the operation is both legal and profitable. Profitability is determined by a heuristic.

The COALESCE_LOOP pragma explicitly enables coalescing of the nested loop construct that follows the pragma. The pragma can only be applied to loops and must appear immediately before a loop construct in C/C++.

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

```
#pragma COALESCE_LOOP
```

If the compiler cannot guarantee that an inner loop is executed at least one time, then it will not coalesce the loop, even if you have used the COALESCE_LOOP pragma. To inform the compiler that a loop will always be executed, use a MUST_ITERATE pragma just prior to the loop body. For example, the following pragma tells the compiler that the loop executes at least once and no more than 65,535 times.

```
#pragma MUST_ITERATE(1,65535,)
```

5.8.4 The CODE_ALIGN Pragma

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

The CODE_ALIGN pragma has the same effect as using the GCC-style `aligned` function attribute. See Section 5.13.2.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

```
#pragma CODE_ALIGN ( constant )
```

5.8.5 The CODE_SECTION Pragma

The CODE_SECTION pragma allocates space for the *symbol* in C, or the next symbol declared in C++, in a section named *section name*. The CODE_SECTION pragma is useful if you have code objects that you want to link into an area separate from the .text section. The CODE_SECTION pragma has the same effect as using the GCC-style `section` function attribute. See Section 5.13.2.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

The following example demonstrates the use of the CODE_SECTION pragma.

Using the CODE_SECTION Pragma in C

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


Copyright © 2026 Texas Instruments Incorporated

5.8.6 The DATA_ALIGN Pragma

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

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

Using the DATA_ALIGN pragma has the same effect as using the GCC-style `aligned` variable attribute. See Section 5.13.4.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

5.8.7 The DATA_MEM_BANK Pragma

The DATA_MEM_BANK pragma aligns a symbol or variable to a specified internal data memory bank boundary. The *constant* specifies a specific memory bank to start your variables on. C7000 devices contain 16 64-bit memory banks. The value of *constant* can be 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, or 15.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma DATA_MEM_BANK (*constant* **)**

Only global variables can be aligned with the DATA_MEM_BANK pragma.

The DATA_MEM_BANK pragma allows you to align data on any data memory bank that can hold data of the type size of the *symbol*. This is useful if you need to align data in a particular way to avoid memory bank conflicts.

This pragma increases the amount of space used in data memory by a small amount as padding is used to align data onto the correct bank.

A value of 'n' for the constant argument to the DATA_MEM_BANK pragma causes the last seven bits of the starting address to be a value equal to 'n*8'. For example, for a value of 1, the last seven bits of the starting address will be 0x08. For a value of 14, the last seven bits of the starting address will be 0x70.

Copyright © 2026 Texas Instruments Incorporated

The code in Example 5-2 uses the DATA_MEM_BANK pragma to specify the alignment of the x, z, w, and zz arrays. It then assigns values to all the array elements and prints the starting address of each array.

Example 5-2. Using the DATA_MEM_BANK Pragma

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

Sample output is as follows:

```
address of w:  0x0000cba0
address of x:  0x0000c610
address of z:  0x0000cc80
address of zz: 0x0000cab0
```

5.8.8 The DATA_SECTION Pragma

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

Using the DATA_SECTION pragma has the same effect as using the GCC-style `section` variable attribute. See Section 5.13.4.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

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

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

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


Copyright © 2026 Texas Instruments Incorporated

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

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

5.8.9 The Diagnostic Message Pragmas

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

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

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

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

Notice that the names of these pragmas are in lowercase.

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

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

Copyright © 2026 Texas Instruments Incorporated

5.8.10 The FORCEINLINE Pragma

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

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

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

```
#pragma FORCEINLINE
```

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

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

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

For information about interactions between command-line options, pragmas, and keywords that affect inlining, see Section 3.10.

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

5.8.11 The FORCEINLINE_RECURSIVE Pragma

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

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

```
#pragma FORCEINLINE_RECURSIVE
```

For information about interactions between command-line options, pragmas, and keywords that affect inlining, see Section 3.10.

Copyright © 2026 Texas Instruments Incorporated

5.8.12 The FUNC_ALWAYS_INLINE Pragma

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

The compiler only inlines a function if it is legal to inline the function. Functions are never inlined if the compiler is invoked with the --opt_level=off option. A function can be inlined even if the function is not declared with the inline keyword. A function can be inlined even if the compiler is not invoked with any --opt_level command-line option. See Section 3.10 for details about interaction between various types of inlining.

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

The FUNC_ALWAYS_INLINE pragma has the same effect as using the GCC-style `always_inline` function attribute. See Section 5.13.2.

The syntax of the pragma in C is:

```
#pragma FUNC_ALWAYS_INLINE ( func )
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_ALWAYS_INLINE
```

The following example uses this pragma:

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

Note

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

5.8.13 The FUNC_CANNOT_INLINE Pragma

The FUNC_CANNOT_INLINE pragma instructs the compiler that the named function cannot be expanded inline. Any function named with this pragma overrides any inlining you designate in any other way, such as using the inline keyword. Automatic inlining is also overridden with this pragma; see Section 3.10.

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

The FUNC_CANNOT_INLINE pragma has the same effect as using the GCC-style `noinline` function attribute. See Section 5.13.2.

The syntax of the pragma in C is:

```
#pragma FUNC_CANNOT_INLINE ( func )
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_CANNOT_INLINE
```


Copyright © 2026 Texas Instruments Incorporated

5.8.14 The FUNC_EXT_CALLED Pragma

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

The FUNC_EXT_CALLED pragma specifies that the optimizer should keep these C functions or any functions these C/C++ functions call. These functions act as entry points into C/C++. The pragma must appear before any declaration or reference to the function to keep. In C, the argument *func* is the name of the function to keep. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

```
#pragma FUNC_EXT_CALLED
```

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

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

5.8.15 The FUNC_IS_PURE Pragma

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

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

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

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

```
#pragma FUNC_IS_PURE
```

5.8.16 The FUNC_IS_SYSTEM Pragma

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

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

The syntax of the pragma in C is:

```
#pragma FUNC_IS_SYSTEM (func )
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_IS_SYSTEM
```


Copyright © 2026 Texas Instruments Incorporated

5.8.17 The FUNC_NEVER_RETURNS Pragma

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

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

The syntax of the pragma in C is:

```
#pragma FUNC_NEVER_RETURNS ( func )
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_NEVER_RETURNS
```

5.8.18 The FUNC_NO_GLOBAL_ASG Pragma

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

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

The syntax of the pragma in C is:

```
#pragma FUNC_NO_GLOBAL_ASG ( func )
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_NO_GLOBAL_ASG
```

5.8.19 The FUNC_NO_IND_ASG Pragma

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

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

The syntax of the pragma in C is:

```
#pragma FUNC_NO_IND_ASG (func )
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_NO_IND_ASG
```

5.8.20 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" )
```


Copyright © 2026 Texas Instruments Incorporated

The syntax of the pragma in C++ is:

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

Supported options for this pragma are --opt_level, --auto_inline, --code_state, --auto_stream, --assume_addresses_ok_for_stream, --diag_suppress, and --opt_for_speed.

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

The following example uses this pragma to set optimization levels for a single function using C:

```
#pragma FUNCTION_OPTIONS(myfunc, "--opt_level=3 --opt_for_speed=4")
```

The following example uses this pragma to set optimization levels for a single function using C++:

```
#pragma FUNCTION_OPTIONS("--opt_level=3 --opt_for_speed=4")
myfunc {
    ...
}
```

5.8.21 The INTERRUPT Pragma

The INTERRUPT pragma enables you to handle interrupts directly with C code.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma INTERRUPT
void *func***(** void **)**

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

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

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

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

Note

Hwi Objects and the INTERRUPT Pragma: The INTERRUPT pragma must not be used when SYS/BIOS Hwi objects are used in conjunction with C functions. The Hwi_enter/Hwi_exit macros and the Hwi dispatcher contain this functionality, and the use of the C modifier can cause negative results.

Copyright © 2026 Texas Instruments Incorporated

5.8.22 The LOCATION Pragma

The compiler supports the ability to specify the run-time address of a variable at the source level. This can be accomplished with the LOCATION pragma or the GCC-style location attribute. The LOCATION pragma has the same effect as using the GCC-style `location` function attribute. See Section 5.13.2.

The syntax of the pragma in C is:

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

The syntax of the pragmas in C++ is:

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

The syntax of the GCC-style attribute (see Section 5.13.4) is:

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

The NOINIT pragma may be used in conjunction with the LOCATION pragma to map variables to special memory locations; see Section 5.8.24.

5.8.23 The MUST_ITERATE Pragma

The MUST_ITERATE pragma specifies to the compiler certain properties of a loop. When you use this pragma, you are guaranteeing to the compiler that a loop executes a specific number of times or a number of times within a specified range.

Any time the UNROLL pragma is applied to a loop, MUST_ITERATE should be applied to the same loop. For loops the MUST_ITERATE pragma's third argument, multiple, is the most important and should always be specified.

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

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

5.8.23.1 The MUST_ITERATE Pragma Syntax

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

```
#pragma MUST_ITERATE (min, max, multiple )
```

The C++ syntax for the corresponding attribute is as follows. No C attribute syntax is available.

```
[[TI::must_iterate(min, max, multiple )]]
```

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

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


Copyright © 2026 Texas Instruments Incorporated

However, if the iteration 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 iteration count is not evenly divisible by multiple. Also, results of the program are undefined if the iteration count is less than the minimum or greater than the maximum specified.

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

The following example uses the must_iterate C++ attribute syntax:

```
void myFunc (int *a, int *b, int * restrict c, int n)
{
    ...
    [[TI::must_iterate(32, 1024, 16)]]
    for (int i = 0; i < n; i++)
    {
        c[i] = a[i] + b[i];
    }
    ...
}
```

5.8.23.2 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 < iteration_count; i++)  { ...
```

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

MUST_ITERATE can specify a range for the iteration count as well as a factor of the iteration count. The following example tells the compiler that the loop executes between 8 and 48 times and the iteration_count variable is a multiple of 8 (8, 16, 24, 32, 40, 48). The multiple argument allows the compiler to unroll the loop.

```
#pragma MUST_ITERATE(8, 48, 8)
for(i = 0; i < iteration_count; i++)  { ...
```

You should consider using MUST_ITERATE for loops with complicated bounds. In the following example, the compiler would have to generate a divide function call to determine, at run time, the number of iterations performed.

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

The compiler will not do the above. In this case, using MUST_ITERATE to specify that the loop always executes eight times allows the compiler to attempt to generate a software pipelined loop:

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

Typically, if the MUST_ITERATE pragma is used to optimize loop execution, a DINT instruction is prepended to the optimized code, the loop code is executed, and an RINT instruction is executed when the loop is terminated.

Copyright © 2026 Texas Instruments Incorporated

5.8.24 The NOINIT and PERSISTENT Pragmas

Global and static variables are zero-initialized by default. However, in applications that use non-volatile memory, it may be desirable to have variables that are not initialized. Noinit variables are global or static variables that are not zero-initialized at startup or reset.

Variables can be declared as noinit or persistent using either pragmas or variable attributes. See Section 5.13.4 for information about using variable attributes in declarations.

Noinit and persistent variables behave identically with the exception of whether or not they are initialized at load time.

- The NOINIT pragma may be used only with uninitialized variables. It prevents such variables from being set to 0 during a reset. It may be used in conjunction with the LOCATION pragma to map variables to special memory locations, like memory-mapped registers, without generating unwanted writes.
- The PERSISTENT pragma may be used only with statically-initialized variables. It prevents such variables from being initialized during a reset. Persistent variables disable startup initialization; they are given an initial value when the code is loaded, but are never again initialized.

By default, noinit or persistent variables are placed in sections named `.TI.noinit` and `.TI.persistent` , respectively. The location of these sections is controlled by the linker command file. Typically .TI.persistent sections are placed in FRAM for devices that support FRAM and .TI.noinit sections are placed in RAM.

Note

When using these pragmas in non-volatile FRAM memory, the memory region could be protected against unintended writes through the device's Memory Protection Unit. Some devices have memory protection enabled by default. Please see the information about memory protection in the datasheet for your device. If the Memory Protection Unit is enabled, it first needs to be disabled before modifying the variables.

If you are using non-volatile RAM, you can define a persistent variable with an initial value of zero loaded into RAM. The program can increment that variable over time as a counter, and that count will not disappear if the device loses power and restarts, because the memory is non-volatile and the boot routines do not initialize it back to zero. For example:

```
#pragma PERSISTENT(x)
#pragma location = 0xC200   // memory address in RAM
int x = 0;
void main() {
    run_init();
    while (1) {
        run_actions(x);
        __delay_cycles(1000000);
        x++;
    }
}
```

The syntax of the pragmas in C is:

#pragma NOINIT (*x*) **int*x*;**
#pragma PERSISTENT (*x*) **int *x*=10;**

Copyright © 2026 Texas Instruments Incorporated

The syntax of the pragmas in C++ is:

```
#pragma NOINIT
intx;
```

```
#pragma PERSISTENT
int x=10;
```

The syntax of the GCC attributes is:

```
int x __attribute__((noinit));
```

```
intx __attribute__((persistent)) = 0;
```

5.8.25 The NOINLINE Pragma

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

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

```
#pragma NOINLINE
```

For information about interactions between command-line options, pragmas, and keywords that affect inlining, see Section 3.10.

5.8.26 The NO_COALESCE_LOOP Pragma

In order to facilitate nested loop coalescing, in which an outer loop level containing instructions is merged into an inner loop level, the C7000 architecture provides a feature known as the Nested Loop Controller (NLC). This feature implements hardware loop control for no more than one level of loop nesting. This allows the compiler to coalesce loop levels while predicating the execution of outer loop instructions in the inner loop. This reduces loop control overhead and provides for better function unit resource utilization for software pipelined loops. Loop coalescing is something that the compiler attempts to do automatically if the operation is both legal and profitable. Profitability is determined by a heuristic.

The NO_COALESCE_LOOP pragma explicitly disables coalescing of the nested loop construct that follows the pragma. The pragma can only be applied to loops and must appear immediately before a loop construct in C/C++.

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

```
#pragma NO_COALESCE_LOOP
```

5.8.27 The NO_HOOKS Pragma

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

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

```
#pragma NO_HOOKS
```

See Section 3.13 for details on entry and exit hooks.

Copyright © 2026 Texas Instruments Incorporated

5.8.28 The once Pragma

The once pragma tells the C preprocessor to ignore a #include directive if that header file has already been included. For example, this pragma may be used if header files contain definitions, such as struct definitions, that would cause a compilation error if they were executed more than once.

This pragma should be used at the beginning of a header file that should only be included once. For example:

```
// hdr.h
#pragma once
#warn You will only see this message one time
struct foo
{
    int member;
};
```

This pragma is not part of the C or C++ standard, but it is a widely-supported preprocessor directive. Note that this pragma does not protect against the inclusion of a header file with the same contents that has been copied to another directory.

5.8.29 The pack Pragma

The pack pragma can be used to control the alignment of fields within a class, struct, or union type. The syntax of the pragma in C/C++ can be any of the following.

#pragma pack (*n***)**

The above form of the pack pragma affects all class, struct, or union type declarations that follow this pragma in a file. It forces the maximum alignment of each field to be the value specified by *n*. Valid values for *n* are 1, 2, 4, 8, and 16 bytes.

#pragma pack (push, *n***)**
#pragma pack (pop)

The above form of the pack pragma affects only class, struct, and union type declarations between push and pop directives. (A pop directive with no prior push results in a warning diagnostic from the compiler.) The maximum alignment of all fields declared is *n*. Valid values for *n* are 1, 2, 4, 8, and 16 bytes.

#pragma pack (show)

The above form of the pack pragma sends a warning diagnostic to stderr to record the current state of the pack pragma stack. You can use this form while debugging.

For more about packed fields, see Section 5.13.5.

Copyright © 2026 Texas Instruments Incorporated

5.8.30 The PROB_ITERATE Pragma

The PROB_ITERATE pragma specifies to the compiler certain properties of a loop. You assert that these properties are true in the common case. The PROB_ITERATE pragma aids the compiler in choosing the best loops and loop transformations (that is, software pipelining and nested loop transformations). PROB_ITERATE is useful only when the MUST_ITERATE pragma is not used or the PROB_ITERATE parameters are more constraining than the MUST_ITERATE parameters.

No statements are allowed between the PROB_ITERATE pragma and the for, while, or do-while loop to which it applies. However, other pragmas, such as UNROLL and MUST_ITERATE, may appear between the PROB_ITERATE pragma and the loop. The syntax of the pragma for C and C++ is:

#pragma PROB_ITERATE(*min* **,** *max* **)**

The C++ syntax for the corresponding attribute is as follows. No C attribute syntax is available. See Section 5.8.23.1 for an example that uses similar syntax.

[[TI::prob_iterate(*min*, *max***)]]**

Where min and max are the minimum and maximum iteration counts of the loop in the common case. The iteration count is the number of times a loop iterates. Both arguments are optional.

For example, PROB_ITERATE could be applied to a loop that executes for eight iterations in the majority of cases (but sometimes may execute more or less than eight iterations):

```
#pragma PROB_ITERATE(8, 8)
```

If only the minimum expected iteration count is known (say it is 5), the pragma would look like this:

```
#pragma PROB_ITERATE(5)
```

If only the maximum expected iteration count is known (say it is 10), the pragma would look like this:

```
#pragma PROB_ITERATE(, 10)  /* Note the blank field for min */
```

5.8.31 The RETAIN Pragma

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

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

The RETAIN pragma has the same effect as using the `retain` function or variable attribute. See Section 5.13.2 and Section 5.13.4, respectively. See Section 11.2.1 for more information about conditional linking.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

#pragma RETAIN

Copyright © 2026 Texas Instruments Incorporated

5.8.32 The SET_CODE_SECTION and SET_DATA_SECTION Pragmas

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

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

#pragma SET_CODE_SECTION ("*section name*")

#pragma SET_DATA_SECTION ("*section name*")

In the Setting Section With SET_DATA_SECTION Pragma example, x and y are put in the section mydata. To reset the current section to the default used by the compiler, a blank parameter should be passed to the pragma. An easy way to think of the pragma is that it is like applying the CODE_SECTION or DATA_SECTION pragma to all symbols below it.

Setting Section With SET_DATA_SECTION Pragma

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

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

Setting a Section With SET_CODE_SECTION Pragma

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

In the Setting a Section With SET_CODE_SECTION Pragma example, func1 is placed in section func1. If conflicting sections are specified at the declaration and definition, a diagnostic is issued.

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

Overriding SET_DATA_SECTION Setting

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

In the Overriding SET_DATA_SECTION Setting example, x is placed in x_data and y is placed in mydata. No diagnostic is issued for this case.

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

The SET_DATA_SECTION pragma takes precedence over the --gen_data_subsections=on option if it is used.

Copyright © 2026 Texas Instruments Incorporated

5.8.33 The STRUCT_ALIGN Pragma

The STRUCT_ALIGN pragma is similar to DATA_ALIGN, but it can be applied to a structure or union type or typedef. It is inherited by any symbol created from that type. The STRUCT_ALIGN pragma is supported only in C.

The syntax of the pragma is:

#pragma STRUCT_ALIGN(*type* **,** *constant expression* **)**

This pragma guarantees that the alignment of the named type or the base type of the named typedef is at least equal to that of the expression. (The alignment may be greater as required by the compiler.) The alignment must be a power of 2. The *type* must be a type or a typedef name. If a type, it must be either a structure tag or a union tag. If a typedef, its base type must be either a structure tag or a union tag.

Note that while the top-level object of a type (or a typedef of that type) will be aligned as requested, the type will not be padded to the alignment (as is usual for a struct), nor does the alignment propagate to derived types such as arrays and parent structs. If you want to pad a structure or union so that individual elements are also aligned and/or cause the alignment to apply to derived types, use the "aligned" type attribute described in Section 5.13.5.

Since ANSI/ISO C declares that a typedef is simply an alias for a type (i.e. a struct) this pragma can be applied to the struct, the typedef of the struct, or any typedef derived from them, and affects all aliases of the base type.

This example aligns any st_tag structure variables on a page boundary:

```
typedef struct st_tag
{
   int   a;
   short b;
} st_typedef;
#pragma STRUCT_ALIGN (st_tag, 128)
#pragma STRUCT_ALIGN (st_typedef, 128)
```

Any use of STRUCT_ALIGN with a basic type (int, short, float) or a variable results in an error.

5.8.34 The UNROLL Pragma

The UNROLL pragma specifies to the compiler how many times a loop should be unrolled. The UNROLL pragma is useful for helping the compiler utilize SIMD instructions. It is also useful in cases where better utilization of software pipeline resources are needed over a non-unrolled loop.

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

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

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

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

The C++ syntax for the corresponding attribute is as follows. No C attribute syntax is available. See Section 5.8.23.1 for an example that uses similar syntax.

[[TI::unroll(*n* **)]]**

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

Copyright © 2026 Texas Instruments Incorporated

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

The compiler can sometimes obtain this information itself by analyzing the code. However, sometimes the compiler can be overly conservative in its assumptions and therefore generates more code than is necessary when unrolling. This can also lead to not unrolling at all. Furthermore, if the mechanism that determines when the loop should exit is complex, the compiler may not be able to determine these properties of the loop. In these cases, you must tell the compiler the properties of the loop by using the MUST_ITERATE pragma.

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

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

5.8.35 The WEAK Pragma

The WEAK pragma gives weak binding to a symbol.

The syntax of the pragma in C is:

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

The syntax of the pragma in C++ is:

```
#pragma WEAK
```

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

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

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

The WEAK pragma has the same effect as using the `weak` function or variable attribute. See Section 5.13.2 and Section 5.13.4, respectively.

Copyright © 2026 Texas Instruments Incorporated

5.9 The _Pragma Operator

The C7000 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 © 2026 Texas Instruments Incorporated

5.10 Application Binary Interface

An Application Binary Interface (ABI) defines how functions that are written separately, and compiled separately can work together. This involves standardizing the data type representation, register conventions, and function structure and calling conventions. An ABI allows ABI-compliant object files to be linked together, regardless of their source, and allows the resulting executable to run on any system that supports that ABI. It defines linkname generation from C symbol names. It also defines the object file format and the debug format, along with documenting how the system is initialized. In the case of C++, it defines C++ name mangling and exception handling support.

The C7000 compiler and linker support only the Embedded Application Binary Interface (EABI) ABI, which works only with object files that use the ELF object file format and the DWARF debug format.

EABI uses the ELF object file format, which supports modern language features like early template instantiation and export inline functions. TI-specific information on EABI mode is described in Section 6.7.2.

5.11 Object File Symbol Naming Conventions (Linknames)

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

User-defined symbols in C code and in assembly code are stored in the same namespace, which means you are responsible for making sure that your C identifiers do not collide with your assembly code identifiers. You may have identifiers that collide with assembly keywords (for instance, register names); in this case, the compiler automatically uses an escape sequence to prevent the collision. The compiler escapes the identifier with double parallel bars, which instructs the assembler not to treat the identifier as a keyword. You are responsible for making sure that C identifiers do not collide with user-defined assembly code identifiers.

Name mangling encodes the types of the parameters of a function in the linkname for a function. Name mangling only occurs for C++ functions which are not declared 'extern "C"'. Mangling allows function overloading, operator overloading, and type-safe linking. Be aware that the return value of the function is not encoded in the mangled name, as C++ functions cannot be overloaded based on the return value.

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

_func__F *parmcodes*

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

For this simple C++ source file:

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

This is the resulting assembly code:

```
_foo__Fi
```

The linkname of foo is _foo__Fi, indicating that foo is a function that takes a single argument of type int. To aid inspection and debugging, a name demangling utility is provided that demangles names into those found in the original C++ source. See Chapter 14 for more information.

The mangling algorithm follows that described in the Itanium C++ ABI (http://www.codesourcery.com/cxx-abi/abi.html).

`int foo(int i) { }` would be mangled "_Z3fooi"

Copyright © 2026 Texas Instruments Incorporated

5.12 Changing the ANSI/ISO C/C++ Language Mode

The language mode command-line options determine how the compiler interprets your source code. You specify one option to identify which language standard your code follows. You can also specify a separate option to specify how strictly the compiler should expect your code to conform to the standard.

Specify one of the following language options to control the language standard that the compiler expects the source to follow. The options are:

- ANSI/ISO C89 (--c89, default for C files)
- ANSI/ISO C99 (--c99, see Section 5.12.1.)
- ANSI/ISO C11 (--c11, see Section 5.12.2)
- ISO C++14 (--c++14, used for all C++ files, see Section 5.2.)

Use one of the following options to specify how strictly the code conforms to the standard:

- Relaxed ANSI/ISO (--relaxed_ansi or -pr) This is the default.
- Strict ANSI/ISO (--strict_ansi or -ps)

The default is relaxed ANSI/ISO mode. Under relaxed ANSI/ISO mode, the compiler accepts language extensions that could potentially conflict with ANSI/ISO C/C++. Under strict ANSI mode, these language extensions are suppressed so that the compiler will accept all strictly conforming programs. (See Section 5.12.3.)

5.12.1 C99 Support (--c99)

The compiler supports the 1999 standard of C as standardized by the ISO. However, the following list of run-time functions and features are *not* implemented or fully supported:

- inttypes.h
  - wcstoimax() / wcstoumax()
- stdio.h
  - The %e specifier may produce "-0" when "0" is expected by the standard
  - snprintf() does not properly pad with spaces when writing to a wide character array
- stdlib.h
  - vfscanf() / vscanf() / vsscanf() return value on floating point matching failure is incorrect
- wchar.h
  - getws() / fputws()
  - mbrlen()
  - mbsrtowcs()
  - wcscat()
  - wcschr()
  - wcscmp() / wcsncmp()
  - wcscpy() / wcsncpy()
  - wcsftime()
  - wcsrtombs()
  - wcsstr()
  - wcstok()
  - wcsxfrm()
  - Wide character print / scan functions
  - Wide character conversion functions

5.12.2 C11 Support (--c11)

The compiler supports the 2011 standard of C as standardized by the ISO. However, in addition to the list in Section 5.12.1, the following functions and features are *not* implemented or fully supported in C11 mode:

- threads.h
- atomic operations

Copyright © 2026 Texas Instruments Incorporated

5.12.3 Strict ANSI Mode and Relaxed ANSI Mode (--strict_ansi and --relaxed_ansi)

Under relaxed ANSI/ISO mode (the default), the compiler accepts language extensions that could potentially conflict with a strictly conforming ANSI/ISO C/C++ program. Under strict ANSI mode, these language extensions are suppressed so that the compiler will accept all strictly conforming programs.

Use the --strict_ansi option when you know your program is a conforming program and it will not compile in relaxed mode. In this mode, language extensions that conflict with ANSI/ISO C/C++ are disabled and the compiler will emit error messages where the standard requires it to do so. Violations that are considered discretionary by the standard may be emitted as warnings instead.

Examples: The following is strictly conforming C code, but will not be accepted by the compiler in the default relaxed mode. To get the compiler to accept this code, use strict ANSI mode. The compiler will suppress the inline keyword language exception, and inline may then be used as an identifier in the code.

```
int main()
{
    int inline = 0;
    return 0;
}
```

The following is not strictly conforming code. The compiler will not accept this code in strict ANSI mode. To get the compiler to accept it, use relaxed ANSI mode. The compiler will provide the int16 type extension and will accept the code.

```
extern int16 myFunc(void);
int main()
{
    return 0;
}
```

The following code is accepted in all modes. The __int16 type does not conflict with the ANSI/ISO C standard, so it is always available as a language extension.

```
extern __int16 myFunc(void);
int main()
{
    return 0;
}
```

The default mode is relaxed ANSI. This mode can be selected with the --relaxed_ansi (or -pr) option. Relaxed ANSI mode accepts the broadest range of programs. It accepts all TI language extensions, even those which conflict with ANSI/ISO, and ignores some ANSI/ISO violations for which the compiler can do something reasonable. Some GCC language extensions described in Section 5.13 may conflict with strict ANSI/ISO standards, but many do not conflict with the standards.

Copyright © 2026 Texas Instruments Incorporated

5.13 GNU and Clang Language Extensions

The GNU compiler collection (GCC) defines a number of language features not found in the ANSI/ISO C and C++ standards. The definition and examples of these extensions (for GCC version 4.7) can be found at the GNU web site, http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/C-Extensions.html#C-Extensions. Most of these extensions are also available for C++ source code.

The compiler supports the following Clang macro extensions, which are described in the Clang 6 documentation:

- __has_feature (up to tests described for Clang 3.5)
- __has_extension (up to tests described for Clang 3.5)
- __has_include
- __has_include_next
- __has_builtin (see Section 5.13.6)
- __has_attribute

5.13.1 Extensions

Most of the GCC language extensions are available in the TI compiler when compiling in relaxed ANSI mode (--relaxed_ansi).

The extensions that the TI compiler supports are listed in Table 5-9, which is based on the list of extensions found at the GNU web site. The shaded rows describe extensions that are not supported.

Table 5-9. 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 number
Zero length	Zero-length arrays
Variadic macros	Macros with a variable number of arguments
Variable length	Arrays whose length is computed at run time
Empty structures	Structures with no members
Subscripting	Any array can be subscripted, even if it is not an lvalue.
Escaped newlines	Slightly looser rules for escaped newlines
Multi-line strings(1)	String literals with embedded newlines
Pointer arithmetic	Arithmetic on void pointers and function pointers
Initializers	Non-constant initializers
Compound literals	Compound literals give structures, unions, or arrays as values
Designated initializers	Labeling elements of initializers
Cast to union	Casting to union type from any member of the union
Case ranges	'Case 1 ... 9' and such
Mixed declarations	Mixing declarations and code
Function attributes	Declaring that functions have no side effects, or that they can never return

Copyright © 2026 Texas Instruments Incorporated

Table 5-9. GCC Language Extensions (continued)

Extensions	Descriptions
Attribute syntax	Formal syntax for attributes
Function prototypes	Prototype declarations and old-style definitions
C++ comments	C++ comments are recognized.
Dollar signs	A dollar sign is allowed in identifiers.
Character escapes	The character ESC is represented as \e
Variable attributes	Specifying the attributes of variables
Type attributes	Specifying the attributes of types
Alignment	Inquiring about the alignment of a type or variable
Inline	Defining inline functions (as fast as macros)
Assembly labels	Specifying the assembler name to use for a C symbol
Extended asm	Assembler instructions with C operands
Constraints	Constraints for asm operands
Wrapper headers	Wrapper header files can include another version of the header file using #include_next
Alternate keywords	Header files can use __const__ 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 5.13.6)
Vector extensions	Using vector operations (see Section 5.14)
Target built-ins	Built-in functions specific to particular targets
Pragmas	Pragmas accepted by GCC
Unnamed fields	Unnamed struct/union fields within structs/unions
Thread-local	Per-thread variables
Binary constants	Binary constants using the '0b' prefix.

(1) Feature defined for GCC 3.0; definition and examples at http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/C-Extensions.html#C-Extensions

Copyright © 2026 Texas Instruments Incorporated

5.13.2 Function Attributes

The following GCC function attributes are supported:

- alias
- aligned
- always_inline
- calls
- const
- constructor
- deprecated
- format
- format_arg
- malloc
- noinline
- noreturn
- pure
- section
- unused
- used
- visibility
- warn_unused_result
- weak

The following additional TI-specific function attribute is supported:

- retain

For example, this function declaration uses the **alias** attribute to make "my_alias" a function alias for the "myFunc" function:

```
void my_alias() __attribute__((alias("myFunc")));
```

The **aligned** function attribute aligns the function using the specified alignment. The alignment must be a power of 2. This attribute has the same effect as the CODE_ALIGN pragma; see Section 5.8.4.

The **always_inline** function attribute acts the same as the FUNC_ALWAYS_INLINE pragma. See Section 5.8.12

The **calls** attribute has the same effect as the CALLS pragma, which is described in Section 5.8.1.

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

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

The **noinline** function attribute has the same effect as the FUNC_CANNOT_INLINE pragma. See Section 5.8.13

The **retain** attribute has the same effect as the RETAIN pragma (Section 5.8.31). That is, the section that contains the function will not be omitted from conditionally linked output even if it is not referenced elsewhere in the application.

The **section** attribute when used on a function has the same effect as the CODE_SECTION pragma. See Section 5.8.5

The **weak** attribute has the same effect as the WEAK pragma (Section 5.8.35).

Copyright © 2026 Texas Instruments Incorporated

5.13.3 For Loop Attributes

If you are using C++, there are several TI-specific attributes that can be applied to loops. No corresponding syntax is available in C. The following TI-specific attributes have the same function as their corresponding pragmas:

- TI::must_iterate
- TI::prob_iterate
- TI::unroll

See Section 5.8.23.1 for an example that uses a for loop attribute.

5.13.4 Variable Attributes

The following variable attributes are supported:

- aligned
- deprecated
- location
- mode
- noinit
- packed
- persistent
- retain
- section
- transparent_union
- unused
- used
- weak

The **aligned** attribute used on a variable has the same effect as the DATA_ALIGN pragma. See Section 5.8.6.

The **location** attribute has the same effect as the LOCATION pragma. See Section 5.8.22. For example:

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

The **noinit** and **persistent** attributes apply to the ROM initialization model and allow an application to avoid initializing certain global variables during a reset. The alternative RAM initialization model initializes variables only when the image is loaded; no variables are initialized during a reset. See RAM Model vs. ROM Model.

The **noinit** attribute can be used on uninitialized variables; it prevents those variables from being set to 0 during a reset. The **persistent** attribute can be used on initialized variables; it prevents those variables from being initialized during a reset. By default, variables marked noinit or persistent will be placed in sections named `.TI.noinit` and `.TI.persistent` , respectively. The location of these sections is controlled by the linker command file. Typically .TI.persistent sections are placed in FRAM for devices that support FRAM and .TI.noinit sections are placed in RAM. Also see Section 5.8.24.

The **packed** attribute may be applied to individual fields within a struct or union. The packed attribute for structure and union fields is available only when there is hardware support for unaligned accesses.

The **retain** attribute has the same effect as the RETAIN pragma (Section 5.8.31). That is, the section that contains the variable will not be omitted from conditionally linked output even if it is not referenced elsewhere in the application.

The **section** attribute when used on a variable has the same effect as the DATA_SECTION pragma. See Section 5.8.8.

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

The **weak** attribute has the same effect as the WEAK pragma (Section 5.8.35).

Copyright © 2026 Texas Instruments Incorporated

5.13.5 Type Attributes

The following type attributes are supported:

- aligned
- deprecated
- packed
- transparent_union
- unused
- visibility

Using the **aligned** type attribute causes the individual elements of a structure, union, or other type to be padded (as is usual for a struct) as needed to achieve the specified alignment for all elements. In addition, any derived types have the same alignment. For example:

```
struct __attribute__((aligned(32))) myStruct { char c1; int i; char c2; };
```

The **packed** attribute is supported for struct and union types. It is available only for target architectures that have hardware support for unaligned access if the --relaxed_ansi option is used.

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

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

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

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

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

Using "packed" on a bit-field overrides the EABI requirements for bit-fields. For *non-packed bit-fields*, the declared type of a bit-field is used for the container type. For *packed bit-fields*, the smallest integral type (except bool) is used, regardless of the declared type. For *non-packed volatile bit-fields*, the bit-field must be accessed using one access of the same size as the declared type. For *packed volatile bit-fields*, the access must be of the same size as the actual container type, and may not be the same as the declared type; additionally, the actual container might not be aligned, and might span more than one aligned container boundary, so accessing a packed volatile bit-field may require more than one memory access. This can also affect the overall size of the struct; for instance, if the struct contains only the bit-field, the struct might not be as large as the declared type of the bit-field. For *both packed and unpacked bit-fields*, bit-fields are bit-aligned and are packed together with adjacent bit-fields with no padding, are entirely contained within an integer container that is at least byte-aligned; and do not alter the alignment of adjacent non-bitfield struct members.

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

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

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


Copyright © 2026 Texas Instruments Incorporated

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

```
void foo(int *param);
struct packed_struct ps;
int *p1 = &ps.i;
int *p2 = (int *)&ps.i;
foo(&ps.i);
```

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

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

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

5.13.6 Built-In Functions

The following built-in functions are supported:

- __builtin_abs()
- __builtin_constant_p()
- __builtin_expect()
- __builtin_fabs()
- __builtin_fabsf()
- __builtin_frame_address()
- __builtin_labs()
- __builtin_memcpy()
- __builtin_return_address()

The __builtin_frame_address() function always returns zero.

The __builtin_return_address() function always returns zero.

Copyright © 2026 Texas Instruments Incorporated

5.14 Operations and Functions for Vector Data Types

The C/C++ compiler supports the use of TI vector data types in C/C++ source files. Vector data types are described in Section 5.3.2. These vector data types are useful for parallel programming applications. Vector data types are enabled by default. You can disable the vector data type names that are *not* prefixed with a double-underscore using the --vectypes compiler option.

Various types of operations can be performed on vectors. These include vector literals and concatenation (Section 5.14.1), unary and binary operators (Section 5.14.2), and swizzle operators for component access (Section 5.14.4).

Note

Vectors cannot be passed to variadic functions (stdarg.h) and cannot be passed to printf().

5.14.1 Vector Literals and Concatenation

You can specify the values to use for vector initialization or assignment using literals or scalar variables. When all of the components assigned to a vector are constants, the result is a vector literal. Otherwise, the vector's value is determined at run-time. When initializing a vector where each element does not have the same initial value, the constructor initialization idiom should be used as in the example below.

For example, the values assigned to vec_a and vec_b in the following declarations are vector literals and are known during compilation:

```
short4 vec_a = short4(1, 2, 3, 4);
float2 vec_b = float2(3.2, -2.3);
```

The following statements initialize all elements of a vector to the same value, which is 1 in this case. In this case, either the "constructor" idiom or the "cast/scalar-widening" idiom can be used.

```
ushort4 myushort4a = ushort4(1); // constructor syntax (preferred)
ushort4 myushort4b = (ushort4)1; // cast/scalar-widening syntax
```

You can initialize TI complex values in a similar manner by specifying the real component first:

```
cfloat rotate_90 = cfloat( /*real=*/0, /*imag=*/1 );
```

Shorter vectors can be concatenated together to form longer vectors. In the following example, two int variables are concatenated into an int2 variable. The value of myvec in the following function is not resolved until run-time:

```
void foo(int a, int b)
{
    int2 myvec = int2(a, b);
    . . .
}
```

This example concatenates two int2 variables into an int4 variable, which is passed to an external function:

```
extern void bar(int4 v4);
void foo(int a, int b)
{
    int2 myv2_a = int2(a, 1);
    int2 myv2_b = int2(b, 2);
    int4 myv4 = int4(myv2_a, myv2_b);
    bar(myv4);
}
```

While vectors of `fp8_e4m3_` and `bf16_` elements can be constructed using this vector syntax, literals are not supported with these types. A vector constructor for these types may need to use `__as_fp8_e4m3()` or `__as_bf16()` syntax to initialize the elements. However, that syntax simply reinterprets the bits/bytes; it does not

Copyright © 2026 Texas Instruments Incorporated

convert the value numerically. Also, that syntax requires the type of the parameter to have the size (number of bits) as the type being converted into, so an `__as_fp8_e4m3()` conversion would require an 8-bit input, and a `__as_bf16()` conversion would require a 16-bit input.

5.14.2 Unary and Binary Operators for Vectors

When an arithmetic operator is applied to a vector, the operator is applied to each element in the vector in parallel. Each element in the resulting vector is the result of applying the operator to the corresponding elements in the source vector(s).

Note

Currently only the assignment operator (=) is supported for bf16_ vectors and fp8_e4m3_ vectors.

The following tables list operations supported for vectors. The "Legal For" column indicates whether the operator can be used with integer vectors, float vectors, complex vectors, Boolean vectors, or multiple vector types.

Table 5-10. Unary and Postfix Operators Supported for Vector Types

Operator	Description	Legal For
-	negate	integer, float, complex
+	returns value unchanged (does not make the value positive)	integer, float, complex
~	bitwise complement	integer
!	logical not	Boolean
&	address of a vector object, which is a pointer to the vector (see Section 5.14.2.1 for examples)	all
++, --	increment / decrement operators (prefix and postfix)	integer, complex integer (not complex float)(1)

(1) Legal for integer vectors, complex vectors, or complex scalars. For complex vectors, only the real component is incremented/ decremented; there is no change to the imaginary component.

The following example declares an int4 vector called pos_i4 and initializes it to the values 1, 2, 3, and 4. It then uses the negate operator to initialize the values of another int4 vector, neg_i4, to the values -1, -2, -3, and -4.

```
int4 pos_i4 = int4(1, 2, 3, 4);
int4 neg_i4 = -pos_i4;
```

Table 5-11. Binary Operators Supported for Vector Types

Operator	Description	Legal For
+, - , *, /	arithmetic operators	integer, float, complex
=	assignment operator	all
+=, -=, *=, /=,	compound assignment operators	integer, float, complex
%	modulo operator	integer
&, |, ^	bitwise logical operators	integer, Boolean
<<, >>	bitwise shift operators	integer
&=, |=, ^=	bitwise compound logical assignment operators	integer, Boolean
<<=, >>=	bitwise compound shift assignment operators	integer
>, >=, ==, !=, <=, <	relational operators	integer, float, Boolean
&&, ||	logical operators. Note that &&= and ||= are not supported.	Boolean

Unlike operations on scalar values, operations on vectors do not perform the usual arithmetic conversions. For example, if you add two vectors of type char, the result is a vector of char, not a vector of int.

For all operators, each input vector must have the same element count and type as other inputs. Resulting vectors have the same element count as the inputs. For arithmetic operators, the result has the same type as

Copyright © 2026 Texas Instruments Incorporated

the input vectors. For logical and comparison operators, the result is a Boolean vector with the same number of elements.

The following example uses the =, ++, and + operators on vectors of type int4. Assume that the iv4 argument initially contains (1, 2, 3, 4). On exit from foo(), iv4 will contain (3, 4, 5, 6).

```
void foo(int4 iv4)
{
    int4 local_iva = iv4++;                 /* local_iva = (1, 2, 3, 4) */
    int4 local_ivb = iv4++;                 /* local_ivb = (2, 3, 4, 5) */

    int4 local_ivc = local_iva + local_ivb; /* local_ivc = (3, 5, 7, 9) */
}
```

The arithmetic operators and increment / decrement operators can be used with complex integral vector types. The increment / decrement operators add or subtract by 1+0i.

The following example multiplies and divides complex vectors of type cfloat2. For details about the rules for complex multiplication and division, please see Annex G of the C99 C language specification.

```
void foo()
{
    cfloat2 va = cfloat2 (1.0, -2.0,  3.0, -4.0);
    cfloat2 vb = cfloat2 (4.0, -2.0, -4.0,  2.0);
   /* vc = (0.0, -10.0), (-4.0, 22.0) */
    cfloat2 vc = va * vb;
   /* vd = (0.4, -0.3), (-1.0, 0.5) */
    cfloat2 vd = va / vb;
    ...
}
```

5.14.2.1 Using Address Operator on a Vector

Vector data object: If you use the unary & operator to take the address of a vector data object, the result is a pointer. Unlike taking the address of an array, taking the address of a vector gives you a pointer representing the whole vector, not a pointer to an individual element.

For example, given an object `vec` of type `int4`, the expression `&vec` has type `int4 *`.

```
int4 vec;
randomize(&vec); /* OK */
```

To access the elements of a pointer vector object, use swizzle operators instead of trying to cast the pointer as some other pointer type. See Section 5.14.4.

```
void randomize(int4 *vecp)
{
    for([..]) (*vecp).s[i] = rand();
}
```

Complex data object: Similarly, if you use the unary & operator to take the address of a complex scalar object, the result is a pointer. This pointer represents the entire complex scalar object, not an individual component.

```
cfloat cplx;
foo(&cplx); /* OK */
```

Vector data element: You can use the unary & operator to take the address of an indexing-style swizzle (for example, `&x.s[1]` or `x.s[j]`). The address is a pointer to that element, with the expected semantics. However, you cannot use the unary & operator to take the address of a non-indexing-style swizzle, such as s1, s1(), or r().

Types with the const qualifier: You can declare vector types and complex types with the const type qualifier. The semantics are the same as for non-vector types. Using the unary & operator to take the address of a const vector object gives a pointer to a const-qualified vector type.

Copyright © 2026 Texas Instruments Incorporated

Types with the volatile qualifier: You can declare vector types and complex types with the volatile type qualifier. The semantics are the same as for non-vector types. Using the unary & operator to take the address of a volatile vector object gives a pointer to a volatile-qualified vector type.

When optimizing access to any volatile object, the compiler must preserve the number and relative order in which accesses occur at runtime. If possible, the compiler accesses a volatile vector object whole-vector-at-a-time, not element-by-element. If that is not possible (for instance, because a vector is larger than a single vector register), the compiler can split the volatile access into smaller chunks, and the order in which these smaller chunks are accessed is unspecified.

Consider these loops. The first loop is the original, and the second loop is the vectorized version:

```
volatile int *p = array;                    /* original loop */
for ([..])
    *p++ = [..]                             /* process one int at a time in sequence */

volatile int8 *p8 = (volatile int8 *)array; /* vectorized loop */
for ([..])
    *p8++ = [..]                            /* process 8 int at a time in parallel */
```

The first loop in this example accesses one array element at a time in sequence. The second loop accesses 8 ints in parallel. Because the second loop changes the relative order of accesses, the compiler is not allowed to automatically convert the first loop to the second. You can manually make a change from one style of access to the other if that change is warranted.

Pointers with the restrict qualifier: You can add the restrict qualifier to a pointer to a vector type if the pointer follows the usual rules for the restrict keyword. Simply stated, the pointer to the vector must be the only way that the vector the pointer points to is accessed.

Correspondence between vectors and arrays: The most effective use of vectors is to process a large array of data with vectors that are as large as possible. Because vectors and arrays are so similar, you can easily use vector pointers to access large arrays. For example:

```
int array[N];
int8 vec = *(int8 *)array; /* vector to access first 8 ints in array */
```

One important use of vectors is to access an array by casting the address of the array (or a pointer) as a pointer-to-vector type. That pointer can then be used to load or store multiple elements of the array in parallel. Suppose you have the following example:

```
void dotp(int x[N], int y[N], int z[restrict])
{
    /* original */
    for (int i=0;i<N;i++)
    {
        /* process the data one int at a time */
        *z++ = *x++ + *y++;
    }
}
```


Copyright © 2026 Texas Instruments Incorporated

The compiler can automatically transform this simple example because the restrict keyword tells the compiler that the z input parameter does not overlap with either x or y.

```
void dotp(int x[N], int y[N], int z[restrict])
{
    int8 *xp = (int8 *)x;
    int8 *yp = (int8 *)y;
    int8 *zp = (int8 *)z;
    for (int i=0;i<N/8;i++)
    {
        /* process the data 8 ints at a time */
        *zp++ = *xp++ + *yp++;
    }
}
```

Note

When using a pointer to a vector to read from a non-complex array, match the element type of the vector to the element type of the non-complex array. For example, if an array type is `const int array[N]`, make the vector a const with int elements, like `const int8`.

Using a pointer-to-vector to access an array is safe because elements of both arrays and vectors are stored and aligned the same way when stored in memory. In memory, a vector's first element (s0) is stored at the lowest address in memory, regardless of endian mode. The individual bytes of each element are stored according to the endianness mode. The first element of the array is then assigned to the vector's first element (s0).

When using C++, you can use similar code to process multiple elements of a std::vector or std::array in parallel using the data() member:

```
void dotp(std::vector<int> x, std::vector<int> y, std::vector<int> z)
{
    int8 *xp = (int8 *)x.data();
    int8 *yp = (int8 *)y.data();
    int8 *zp = (int8 *)z.data();
    for (int i=0;i<N/8;i++)
    {
        /* process the data 8 ints at a time */
        *zp++ = *xp++ + *yp++;
    }
}

void dotp(std::array<int> x, std::array<int> y, std::array<int> z)
{
    int8 *xp = (int8 *)x.data();
    int8 *yp = (int8 *)y.data();
    int8 *zp = (int8 *)z.data();
    for (int i=0;i<N/8;i++)
    {
        /* process the data 8 ints at a time */
        *zp++ = *xp++ + *yp++;
    }
}
```

Accessing a complex type as a vector: The imaginary and real components of complex vector objects and complex scalar objects are stored differently depending on whether the object is in memory or in registers.

Complex scalar objects are stored in memory with the real component at the lowest address. Complex vector objects are stored in memory as a sequence of complex scalar values, with the real component of s0 at the lowest address, followed by the imaginary component of s0, and so on.

Complex scalar objects are stored in registers with the imaginary component in the least significant bit, followed by the real component. Complex vector objects are stored as a sequence of complex scalar values, with imaginary component of s0 at the LSB, followed by real component of s0, and so on. This register layout is required by CPU instructions that operate on complex values (for example, VCMPYSP).

As a result of the reversal in the order in which the components are stored, the compiler must swap the complex real and imaginary components when loading from memory to a register and vice versa.

Copyright © 2026 Texas Instruments Incorporated

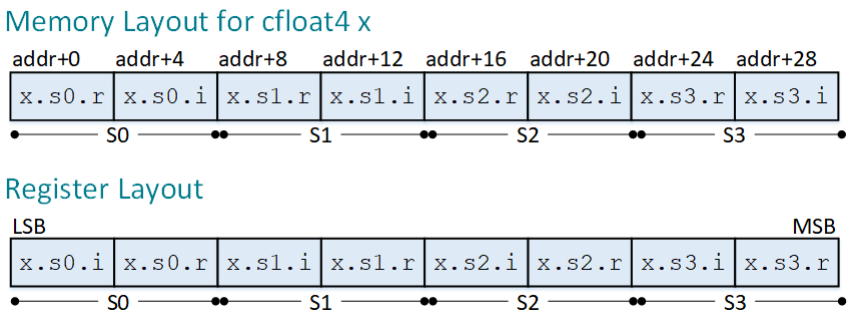


Figure 5-1. Memory and Register Layout of Complex Components

The compiler normally handles swapping the order of complex components. When you access a complex scalar or complex vector object through a pointer, the pointer must have the correct type.

You can access a non-complex array as a complex scalar:

```
float value[2];
cfloat x = *(cfloat *)&value;
```

You can access a complex scalar as a vector of length two with the same element type as the complex components:

```
cchar value;
char2 x = *(char2 *)&cchar;
```

You can access an array of non-complex scalars as a complex vector as long as the complex values are stored in the non-complex scalar array with the real component first, then the imaginary component. You can use either a 1- or 2-dimensional array, as convenient.

```
float value[2][] = { { r, i }, { r, i } ... };
cfloat4 x = *(cfloat4 *)&value;

float value[] = { r, i, r, i, ... };
cfloat4 x = *(cfloat4 *)&value;
```

You can access any complex vector as a vector of twice the length with the same element type as the complex components:

```
cchar4 value;
char8 x = *(char8 *)&value;
```

When loading a complex scalar or complex vector from a non-complex type, the compiler handles the necessary component swapping for you.

Note

When using a pointer to complex to read from a non-complex array, make the element type of the complex object match the element type of the non-complex array. For example, if your array is of type `const int coefficients[2][N]`, make the vector a const with int elements (for example, `const cint8`). If you do not stick to this type, undesirable behavior can occur due to automatic swapping of complex components.

Copyright © 2026 Texas Instruments Incorporated

A frequent use case for complex values in C code is to represent them as two-dimensional array of the component values, like so:

```
float coefficients[2][4] =
{ { real, imag }, { real, imag },
  { real, imag }, { real, imag }, };

/* access part of the coefficient array as a vector */
output = input * *(cfloat4 *)coefficients;
```

You can also initialize a complex vector from an array of C99 complex values, because the layout of a C99 complex object and a TI complex object are the same:

```
float _Complex coefficients[4] =
{ real + imag * _Complex_I, real + imag * _Complex_I,
  real + imag * _Complex_I, real + imag * _Complex_I, };

/* access part of the coefficient array as a vector */
output = input * *(cfloat4 *)coefficients;
```

Mismatched vector/array sizes: You can use vector operations even if an array has an element count that is not an even multiple of the vector element count by using predicated loads and stores for the last iteration of the loop. As a result, the last iteration of the loop loads and stores only part of the vector, and thus only the remaining part of the array.

```
#include <stddef.h>
#include <c7x.h>

void dotp(int x[], int y[], int z[restrict], size_t n)
{
    int8 *xp = (int8 *)x;
    int8 *yp = (int8 *)y;
    int8 *zp = (int8 *)z;
    for (int i=0;i<n/8;i++)
        *zp++ = *xp++ * *yp++;
    unsigned leftover = n%8;
    if (leftover)
    {
        vpred mask = __mask_char(leftover);
        __vstore_pred(mask, zp, __vload_pred(mask, xp) * __vload_pred(mask, yp));
    }
}
```

5.14.3 Ternary Operators for Vectors (?:)

Ternary operators are supported for vector types by the C7000 compiler. Ternary operators require Boolean vectors as the conditional argument and can have any vector type for the second and third arguments. Ternary operators are equivalent to a vector select intrinsic, and therefore all expressions are evaluated.

5.14.4 Swizzle Operators for Vectors

The programming model implementation supports the following "swizzle" operators. These postfix operators allow you to read or write one or more elements of a vector symbolically. These operators use a syntax that makes the access look like accessing a member of a struct or class. An expression using these operators may be used on either side (left or right) of an assignment operator.

.s0(), .s1(), ..., .s9(), .sa(), ..., .sf() Access any one of the first 16 elements of a vector. The first element of the vector is s0.

```
char16 v = char16(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15);
v.s0() = v.sa() * v.sf(); /* now v=(150, 1, 2, 3, ...) */
```

This syntax cannot support accessing elements beyond the 16th element. To access other elements in vectors with more than 16 elements, use the array-indexing syntax below.

Copyright © 2026 Texas Instruments Incorporated

.s[0], .s[1], ..., .s[N-1]

Access any one single element of a vector, where N is the number of elements in the vector. One benefit of using this array-indexing syntax is that the index may be an integer expression, not just a constant.

```
int8 v;

char16 v = char16(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15);
v.s[0] = v.s[10] * v.s[15]; /* now v=(150, 1, 2, 3, ...) */

for (int i=0; i<8; i++) {
   printf("%d\n", v.s[i]);
}
```

If you use this array-indexing style syntax and the index expression is not a constant, you cannot use this operator with any other swizzle operator expression. For example:

```
int8 v;
v.even().s[i] // disallowed
```

.x(), .y(), .z(), or .w()

Access any one of the first 4 elements of a vector. The .x() .y() z.() and .w() operators are aliases for .s0() .s1() .s2() and .s3() respectively.

```
char2 c = char2(0, 1); // 2-D coordinates
int x = c.x(); // x=0
int y = c.y(); // y=1
c.y() = 10; // now c=(0, 10)

char3 c = char3(0, 1, 2); // 3-D coordinates
int x = c.x(); // x=0
int y = c.y(); // y=1
int z = c.z(); // z=2
c.y() = 10; // now c=(0, 10, 2)

char4 c = char4(0, 1, 2, 3); // 4-D coordinates
int x = c.x(); // x=0
int y = c.y(); // y=1
int z = c.z(); // z=2
int w = c.w(); // w=3
c.y() = 10; // now c=(0, 10, 2, 3)
```

.even(), .odd()

Access all the even- or odd-indexed elements of a vector, where the zeroth element is even. If the operand vector has size two, the result is a scalar. If the operand vector has size larger than two, the result is a smaller vector.

```
int4 a;
int2 b = a.even();
int  c = b.even(); // now w=x.s[0]

char8 v = char8(0, 1, 2, 3, 4, 5, 6, 7);
char4 veven = v.even();
v.odd() = veven; /* now v=(0, 0, 2, 2, 4, 4, 6, 6) */
```

You can chain these swizzle operators as long as the operand is a vector or a complex scalar type.

```
char8 w;
w.even().even() = 0;
w.odd().even() = 1;
w.even().odd() = 2;
w.odd().odd() = 3; /* now w=(0, 1, 2, 3, 0, 1, 2, 3) */
```

You may not use .even() or .odd() on vectors of with odd sizes.

.hi(), .lo()

Access all elements in the high- or low-indexed elements of a vector, where the zeroth element is in .lo(). If the operand vector has size two, the result

Copyright © 2026 Texas Instruments Incorporated

is a scalar. If the operand vector has size larger than two, the result is a smaller vector.

```
char8 v = char8(0, 1, 2, 3, 4, 5, 6, 7);
char4 vlo = v.lo();
v.lo() = -vlo; /* now v=(0, -1, -2, -3, 4, 5, 6, 7) */
```

You can chain these swizzle operators as long as the operand is a vector or a complex scalar type.

```
char8 w;
w.lo().lo() = 0;
w.lo().hi() = 1;
w.hi().lo() = 2;
w.hi().hi() = 3; /* now w=(0, 0, 1, 1, 2, 2, 3, 3) */
```

.r(), .i()

Access all the real or imaginary parts of each element of a vector of complex operand, or access the single real or imaginary part of a complex scalar operand. If the operand is a vector of complex, the result is a vector of the same size with scalar elements. If the operand is a complex scalar, the result is a single scalar.

```
cchar c;
char r = c.r();
char i = c.i();

cchar4 v = cchar4(cchar(0, 1), cchar(2, 3), cchar(4, 5), cchar(6,
7));
char4 vr = v.r(); /* now vr=(0, 2, 4, 6) */
char4 vi = v.i(); /* now vr=(1, 3, 5, 7) */

v.r() = vi;
v.i() = vr; /* now v=((1, 0), (3, 2), (5, 4), (7, 6)) */
```

It is undefined behavior to use a swizzle to try to access an element beyond the size of a vector. For example, it is undefined behavior to use `int4.s[4]`.

You may use the unary & operator to take the address of an array-indexing style vector access.

```
int4 v;
cint4 cv;

&v.s[0]    // allowed; this is a pointer to int
&cv.s[idx] // allowed; this is a pointer to cint
```

You may not take the address of any other swizzle expression; use the array-indexing syntax only.

```
int4 v;
&v.s0      // disallowed
```

5.14.5 Unsupported Vector Comparison Operators

Comparisons of vectors as conditions in `if` -statements are not supported by the C7000 compiler. For example if `src1` and `src2` are vectors, the following would be an unsupported use:

```
int ex_compare_unsupported(short16 src1, short16 src2)
{
    if (src1 > src2) { return 1; } // Unsupported, will not work properly
    else             { return 2; }
}
```

You should instead use the vector comparison intrinsics listed in the `c7x.h` runtime support header file to construct a vector predicate. Vector predicates are described in Section 5.14.8.

Copyright © 2026 Texas Instruments Incorporated

5.14.6 Conversion Functions for Vectors

You cannot use standard type casting on vector data types. Instead, convert_<*destination type*>(<*source type*>) functions are provided to convert the elements of one vector type object into another vector type object. This is done on an element-by-element basis, and the source vector type and the destination vector type must be of the same length. That is, 4-element vectors can only be converted to other types of 4-element vectors.

The following example initializes a short2 vector using two ints concatenated to form an int2 vector:

```
void foo(int a, int b)
{
    short2 svec2 = convert_short2(int2(a, b));
    ...
}
```

5.14.7 Re-Interpretation Functions for Vectors

The as_<*destination type*>(<*source type object*>) functions are provided to re-interpret the original type of an object as another vector type. The source type and destination type must be the same size in number of bits. An error is returned if the sizes are different.

Reinterpretations of Boolean vectors that do not result in each element containing exactly 0x0 or 0x1 are undefined. The following shows examples where the result is defined or undefined:

```
ushort2 myshort2_0 = ushort2(0,1);
bool4 mybool4_0 = as_bool4(myshort2_0); // Defined

ushort2 myshort2_1 = ushort2(2,3);
bool4 mybool4_1 = as_bool4(myshort2_1); // Undefined

bool8 mybool8_0 = bool8(0,1,0,1,0,1,0,1);
float2 myfloat2_0 = as_float2(mybool8_0); // Defined

float2 myfloat2_1 = float2(1.0,2.0);
bool8 mybool8_1 = as_bool8(myfloat2_1); // Undefined
```

While arithmetic conversion is performed by the conversion functions described in the previous section, no arithmetic conversion is performed by the re-interpretation functions. For example, suppose a float value of 1.0 is re-interpreted as an int value. Since the float value of 1.0 is represented in hex as 0x3f800000, the value in the resulting int is 1,065,353,216.

The following example reinterprets a non-vector variable of the long type (64 bits) to a float2 vector (2 elements of 32 bits each). The least significant 32 bits of mylong are placed in fltvec2.s0() and the most significant 32 bits of mylong are placed in fltvec2.s1(). No arithmetic conversion is performed.

```
extern long mylong;
float2 fltvec2 = as_float2(mylong);
```

If the sizes of the source and destination types are different, an error occurs.

If vector data types are enabled, you can also use the as_<*type*>() functions for scalar (non-vector) types. The types must have the same number of bits. The following example re-interprets a float value as an int value. Since the float value of 1.0 is represented in hex as 0x3f800000, the value in the resulting int is 1,065,353,216.

```
float myfloat = 1.0f;
myint = as_int(myfloat);
```

5.14.8 Vector Predicate Type

The vector predicate capability is managed through a special, opaque type named `__vpred` . Each bit of a vector predicate value corresponds to one byte lane in a TI vector type. The vector predicate size is 1/8 the size of the vector register for the target. See Table 5-7 for target-specific vector sizes.

Copyright © 2026 Texas Instruments Incorporated

5.14.8.1 Constructing a Vector Predicate Type

Values of type `__vpred` can only be constructed using intrinsics that produce a vector predicate, which are listed in the `c7x_vpred.h` runtime support header file, including:

- Vector Comparison intrinsics: `__cmp_{eq, ge, gt, le, lt}_pred(…)`
  Note: Comparisons between partial vectors (less than the native vector register width in size) will produce a vector predicate value in which the upper, invalid lanes are masked to false.
- Vector Predicate Mask intrinsics: `__mask_{char, short, int, long}`
- Vector Predicate Instructions that operate on other vector predicates: `__negate(__vpred src)`

5.14.8.2 Using a Vector Predicate Type

A constructed vector predicate value of type `__vpred` can be used in any intrinsic shown to accept this value as an operand. Intrinsics that rely on vector predicate types are listed in `c7x_vpred.h`. Other operations on vector predicates—predicated add, predicated subtract, predicated shift, and predicated store—can be performed. Most operations can be made vector predicatable using the vector select `__select(vpred, ...)` intrinsic:

```
int16 ex_compare_and_select(int16 src1, int16 src2)
{
    vpred condition = __cmp_gt_pred(src1, src2);
    int16 result1 = src1 - src2; // if-clause
    int16 result2 = src2 - src1; // else-clause
    return __select(condition, result1, result2);
}
```

Note

Partial element predication is undefined behavior: If a vector predicate is used to control a vector operation, the predicate bits for each element of that operation are assumed by the compiler to be all ones or all zeros. Partially predicating an element of an operation such that the predicate bits for that element of the operation are not all ones or all zeros is considered undefined behavior.

5.14.8.3 Boolean Vector Types

Boolean vectors are a vector data type in C/C++ source files. Vector data types are described in Section 5.3.2. Currently, Boolean vectors allow literals and concatenation (Section 5.14.1), swizzle operators for vectors (Section 5.14.4), conversion functions for vectors (Section 5.14.6), re-interpretation functions for vectors (Section 5.14.7), and the following operators: =,!, &&, ||, &, &=, |, |=, ^, ^=, ==, !=, <=,<, >=,>.

Boolean vectors may be used as an abstract alternative to the low-level vector predicate type (Section 5.14.8) on most predicated intrinsics on the C7000. The use of Boolean vectors as vector predicates is encouraged. However, the Boolean vector type is not fully interchangeable with the low-level vector predicate type.

When using a vector predicate type, you should perform proper scaling on predicates. Vector predicates can be scaled up or down by a factor *k*={0-63} through the intrinsics `__expand_vpred(__vpred, k)` and `__pack_vpred(__vpred, k)`.

Copyright © 2026 Texas Instruments Incorporated

The two functions in the following example achieve the same result, one with a Boolean vector and the other with a low-level vector predicate type. This shows differences in the capabilities of vector predicates vs. Boolean vectors. Boolean vectors predicate input data by lane regardless of the element type.

```
// Boolean vector example
void foo(int4 *ptr, int4 data, char4 *ptr2, char4 data2)
{
    bool4 pred = bool4(0,1,1,0);
    __vstore_pred(pred, ptr, data);   // Word-based store
    __vstore_pred(pred, ptr2, data2); // Byte-based store
}

// Vector predicate example
void bar(int4 *ptr, int4 data, char4 *ptr2, char4 data2)
{
    __vpred pred = _mvrp(0x0000000000000ff0); // Word-scaled predicate
    __vstore_pred(pred, ptr, data);

    pred = __pack_vpred(pred, 2); // Byte-scaled predicate
    __vstore_pred(pred, ptr2, data2);
}
```

5.14.8.4 Legacy Logical and Boolean Vector Operations

Previously, the C7000 compiler's handling of logical operations on Boolean vectors differed from the current default behavior. The legacy logical behavior can be re-enabled with the --deprecated_logical_vector_semantics option. This option may be removed in a future compiler release.

Table 5-12. Legacy Logical and Boolean Vector Operations

Operators	Legacy Behavior	Current Default Behavior
!, &&, ||	Any non-Boolean integral vector is accepted with these operators.	Only Boolean integral vectors are accepted with these operators.
?:	Any non-Boolean integral vector is accepted for the first argument. Boolean vectors are not accepted for the second and third arguments. The behavior is not equivalent to a select intrinsic. Optimized code is not generated.	Only Boolean integral vectors are accepted as the first argument. Any vector type is accepted in the second and third arguments. The behavior is equivalent to a select intrinsic. Optimized code is generated.
&, &=, |, |=, ^, ^=	Boolean vectors are not supported as arguments.	Boolean vectors are supported as arguments.
==,!=,<,<=,>,>=	Boolean vectors are not supported as arguments. The legacy logical behavior as described below is used.	Boolean vectors are supported as arguments.

Legacy logical operations on vectors returned a vector type of the same number of elements as the vector operands with signed integer elements. For example, if an == operator compares two float4 types, the resulting type would be an int4. Comparing two double8 types would result in a long8 type. A vector logical operator results in -1 (for true) or 0 (for false) in each result vector lane.

If you want to use the legacy logical behavior in only a few situations, compile without the --deprecated_logical_vector_semantics option and use the ternary operator (?:) to create the -1 and 0 results as required for those situations.

Copyright © 2026 Texas Instruments Incorporated

5.15 C7000 Intrinsics

The C7000 compiler provides intrinsics to provide functionality for ISA instructions and routines that cannot be leveraged by way of a simple C/C++ operator.

Most instructions defined for the C7000 ISA are available via intrinsics, which are listed in the `c7x.h` runtime-support header file. These intrinsics are subdivided into the following usage categories:

- High-level overloaded intrinsics (Section 5.15.1)
- Intrinsics defined for special load or store instructions (Section 5.15.2)
- Low-level, direct-mapped intrinsics (Section 5.15.3)
- Intrinsics used to perform lookup table and histogram operations (Section 5.15.4)
- Legacy intrinsics used to migrate code written for the C6000 compiler (Section 5.15.6)
- Intrinsics to control the Streaming Engine and Streaming Address Generator (see the Section 4.15)

5.15.1 High-Level Overloaded Intrinsics

The first set of intrinsics provided by the C7000 compiler pertains to operations for which multiple scalar and vector types apply. As such, for each operation to which they apply, these intrinsics have the same name and are overloaded based on the input type(s). Because they are overloaded, these intrinsics are the most abstract and high-level of all of the supported intrinsics.

For example, the C7000 ISA defines a set of absolute value instructions for vectors of bytes (VABSB for up to 64 elements), vectors of half-words (VABSH for up to 32 elements), vectors of words (VABSW for up to 16 elements), vectors of double-words (VABSD for up to 8 elements), vectors of single-precision floating point (VABSSP for up to 16 elements), and vectors of double-precision floating point (VABSDP for up to 8 elements). In spite of this variability, the same intrinsic name is used to access all operations, and the operation is distinguished solely based upon input operand type. This is contained in `c7x.h` as follows:

```
/*---------------------------------------------------------------------------
 * ID: __abs
 *---------------------------------------------------------------------------*/
VABSB
char = __abs(char);
char2 = __abs(char2);
char3 = __abs(char3);
char4 = __abs(char4);
char8 = __abs(char8);
char16 = __abs(char16);
char32 = __abs(char32);
char64 = __abs(char64);
VABSH
short = __abs(short);
short2 = __abs(short2);
short3 = __abs(short3);
short4 = __abs(short4);
short8 = __abs(short8);
short16 = __abs(short16);
short32 = __abs(short32);
VABSW
int = __abs(int);
int2 = __abs(int2);
int3 = __abs(int3);
int4 = __abs(int4);
int8 = __abs(int8);
int16 = __abs(int16);
VABSD
long = __abs(long);
long2 = __abs(long2);
long3 = __abs(long3);
long4 = __abs(long4);
long8 = __abs(long8);
VABSSP
float = __abs(float);
float2 = __abs(float2);
float3 = __abs(float3);
float4 = __abs(float4);
float8 = __abs(float8);
```


Copyright © 2026 Texas Instruments Incorporated

```
float16 = __abs(float16);
VABSDP
double = __abs(double);
double2 = __abs(double2);
double3 = __abs(double3);
double4 = __abs(double4);
double8 = __abs(double8);
```

As long as the intrinsic name is used (`__abs(…)` in this case), the compiler generates the correct corresponding instruction for the input type used. See `c7x.h` for a complete list.

5.15.2 Intrinsics Defined for Special Load and Store Instructions

The C7000 ISA supports several load and store operations that cannot be leveraged using simple C/C++ operators. Instead, an overloaded intrinsic is provided for those operations using prefixes “__vload_” and “__vstore_” for loads and stores, respectively. Overloaded intrinsics are provided for the following load and store operations:

- Vector Load and Duplicate: `__vload_dup(…)`
- Vector Load and Duplicate Group: `__vload_dup_vec(…)`
- Vector Load and Unpack: `__vload_unpack_{short, int, long}(…)`
- Vector Load and Deinterleave: `__vload_deinterleave_{int, long}(...)`
- Vector Interleave and Store: `__vstore_interleave(…)`
- Vector Packing Store: `__vstore_{packl, packh, packhs1, pack_byte}(…)`
- Vector Reverse Bit Store: `__vstore_reverse_bit(…)`
- Vector Predicated Store: `__vstore_pred(vpred, …)`
- Vector Interleave, Predicated Store: `__vstore_pred_interleave(vpred, …)`
- Vector Predicated Packing Store: `__vstore_pred_{packl, packh, packhs1, pack_byte}(vpred, ...)`
- Vector Predicated Reverse Bit Store: `__vstore_pred_reverse_bit(vpred, …)`
- Vector Constant Store: `__vstore_const_{2word, 4word, 8word, 16word}(…)`
- Store of Vector Predicate: `__store_predicate_{char, short, int, long}(…)`
- Atomic Swap: `__atomic_swap(…)`
- Atomic Compare and Swap: `__atomic_compare_swap(…)`

Copyright © 2026 Texas Instruments Incorporated

5.15.3 Direct-Mapped Intrinsics

Intrinsics are also provided to facilitate a direct-map between an intrinsic and a corresponding C7000 instruction. As such, these intrinsics are *not overloaded*; they are therefore the least abstracted from the hardware and are therefore considered *low-level*. The primary purpose of these low-level intrinsics is to ensure that no other instructions are generated other than those desired by the programmer. This is particularly useful for operations that require operand *interleaving* on input or operand *deinterleaving* on output.

All direct-mapped intrinsics are listed in `c7x_direct.h` , which is included by the top-level `c7x.h` file.

For example, the C7000 instruction VCMATMPYHW (vector complex matrix multiply) requires that its second source operand be interleaved along 64-bit boundaries. It also requires that its output also be deinterleaved along 64-bit boundaries. As listed in `c7x.h` and `c7x_direct.h` , the C7000 compiler provides two interfaces into this instruction:

- High-level intrinsic:

```
/*---------------------------------------------------------------------------
 * ID: __cmatmpy_ext
 *---------------------------------------------------------------------------*/
/*
VCMATMPYHW
cint2 = __cmatmpy_ext(cshort2, cshort4);
cint4 = __cmatmpy_ext(cshort4, cshort8);
cint8 = __cmatmpy_ext(cshort8, cshort16);
*/
```

- Low-level intrinsic:

```
/*---------------------------------------------------------------------------
 * ID: __vcmatmpyhw_vww
 *---------------------------------------------------------------------------*/
/*
VCMATMPYHW
__vcmatmpyhw_vww(cshort16, cshort16, cshort16, cint8&, cint8&);
*/
```

If you choose to use the overloaded "__cmatmpy_ext(…)" intrinsic, the compiler will assume that both the input and output data is *not interleaved* and will attempt to abstract this. Therefore, the compiler will insert special instructions to interleave the input prior to instruction execution, and it will also insert special instructions to deinterleave the output after instruction execution. *This method leans toward programmer ease-of-use at the expense of instruction cycles.*

More advanced programmers may instead opt to use the direct-mapped, low-level "__vcmatmpyhw_vww(…)" intrinsic and manage the interleaving and deinterleaving themselves. In this case, the interleaved input is shown as a pair of `cshort16` vectors, and the output is given as a pair of `cint8` vectors, each of which are determined by the maximum width and basic type supported by the VCMATMPYHW instruction.

5.15.4 Lookup Table and Histogram Intrinsics

The intrinsics used to configure and use the C7000 lookup table and histogram features are listed in `c7x_luthist.h` , which is included as part of C7000 runtime support.

5.15.5 Matrix-Multiply Accelerator (MMA) Intrinsics

The intrinsics used to configure and use the Matrix-Multiply Accelerator (MMA) are listed in `c7x_mma.h` , which is included as part of C7000 runtime support.

5.15.6 Legacy Intrinsics

The C7000 runtime support defines C6000 legacy intrinsics that can be used to compile C6000 source code using the C7000 compiler. Compiling source code containing these intrinsics should be done according to information provided in the *C6000-to-C7000 Migration User's Guide* (SPRUIG5) as well as the `c6x_migration.h` runtime support header file.

Copyright © 2026 Texas Instruments Incorporated

5.16 C7000 Scalable Vector Programming

A set of utilities are provided in the compiler library for writing vector-width independent code for C7000. To use these utilities, #include `c7x_scalable.h` in source code. The classes provided by c7x_scalable.h use the TI vector types described in Section 5.3.2 and all the same requirements apply.

These utilities are available for use in C++ code only due to use of C++ language features in the implementation.

These utilities are available when using the TI C7000 compiler or when compiling with TI C7000 Host Emulation.

The following APIs are available, all of which are described in further detail in the `c7x_scalable.h` file:

- Vector type query and construction

```
c7x::max_simd<T>::value
c7x::element_count_of<T>::value
c7x::element_type_of<T>::type
c7x::component_type_of<T>::type
c7x::make_vector<T,N>::type
c7x::make_full_vector<T>::type
c7x::is_target_vector<T>::value
```

- Full vector types

```
c7x::char_vec
 c7x::short_vec
 etc
```

- Half vector types

```
c7x::char_hvec
c7x::short_hvec
etc
```

- Quarter vector types

```
c7x::char_qvec
c7x::short_qvec
etc
```

- Eighth vector types

```
c7x::char_evec
c7x::short_evec
etc
```

- Templated vector reinterprets and conversions

```
c7x::reinterpret<T>(v)
c7x::convert<T>(v)
```

- Vector reinterprets and conversions

```
c7x::as_char_vec(v)
c7x::convert_short_vec(v)etc
```

- Streaming Engine and Streaming Address Generator helpers

```
c7x::se_veclen<T>::value
c7x::se_eletype<T>::value
c7x::sa_veclen<T>::value
c7x::strm_eng<I,T>::get()
c7x::strm_eng<I,T>::get_adv()
c7x::strm_agen<I,T>::get(p)
c7x::strm_agen<I,T>::get_adv(p)
c7x::strm_agen<I,T>::get_vpred()
```

The following macros are defined by `c7x_mma.h` and can be used to determine information about the MMA for use with the scalable vector programming model:

Copyright © 2026 Texas Instruments Incorporated

Table 5-13. Macros for Use with MMA and Scalable Vector Programming

Macro Syntax	Description
__MMA_A_MAT_BYTES__	The size of an A matrix in bytes. Currently, each A matrix contains one row.
__MMA_A_ROW_WIDTH_BYTES__	The size of a row in an A matrix in bytes.
__MMA_A_ROWS__	The number of rows in an A matrix.
__MMA_A_COLS(ebytes)	The number of columns in an A matrix given the number of bytes in each element of A. Often useful with sizeof(). For example, `__MMA_A_COLS(sizeof(short))`.
__MMA_A_ENTRIES__	The number of A entries that can be contained in the A storage.
__MMA_B_MAT_BYTES__	The size of a B matrix in bytes.
__MMA_B_ROW_WIDTH_BYTES__	The size of a row in a B matrix in bytes.
__MMA_B_ROWS(ebytes)	The number of rows in a B matrix given the number of bytes in each element of B. Often useful with sizeof(). For example, `__MMA_B_ROWS(sizeof(short))`.
__MMA_B_COLS(ebytes)	The number of columns in a B matrix given the number of bytes in each element of B. Often useful with sizeof(). For example, `__MMA_B_COLS(sizeof(short))`.
__MMA_C_MAT_BYTES__	The size of a C matrix. Currently, each C matrix contains one row. Currently the C matrix is 4 times wider than the A matrix for larger accumulators.
__MMA_C_ROW_WIDTH_BYTES__	The size of a row in a C matrix.
__MMA_C_ROWS__	The number of rows in a C matrix.
__MMA_C_COLS(ebytes)	The number of columns in a C matrix given the number of bytes in each element of C. Often useful with sizeof(). For example, `__MMA_C_COLS(sizeof(short))`.
__MMA_C_ENTRIES__	The number of C entries that can be contained in C storage.

Copyright © 2026 Texas Instruments Incorporated

As a moderate complexity example, the following is an implementation of a C++ function template for memcpy that uses the input type as a template. This example uses a Streaming Engine and a Streaming Address Generator (see Section 4.15).

```
#include <c7x_scalable.h>

using namespace c7x;

/* memcpy_scalable_strm<typename S>(const S*in, S *out, int len)
 *
 * S   - A basic data type such as short or float.
 * in  - The input buffer.
 * out - The output buffer.
 * len - The number of elements to copy.
 *
 * Defaulted template arguments:
 * V   - A full vector type of S
 */
template<typename S,
         typename V  = typename make_full_vector<S>::type>
void memcpy_scalable_strm(const S *restrict in, S *restrict out, int len)
{
    /*
     * Find the maximum number of vector loads/stores needed to copy the buffer,
     * including any remainder.
     */
    int cnt = len / element_count_of<V>::value;
    cnt += (len % element_count_of<V>::value > 0);

    /* Initialize the SE for a linear read in and the SA for a linear write out. */
    __SE_TEMPLATE_v1 in_tmplt = __gen_SE_TEMPLATE_v1();
    __SA_TEMPLATE_v1 out_tmplt = __gen_SA_TEMPLATE_v1();

    in_tmplt.VECLEN = se_veclen<V>::value;
    in_tmplt.ELETYPE = se_eletype<V>::value;
    in_tmplt.ICNT0 = len;

    out_tmplt.VECLEN = sa_veclen<V>::value;
    out_tmplt.ICNT0 = len;

    __SE0_OPEN(in, in_tmplt);
    __SA0_OPEN(out_tmplt);

    /* Perform the copy. If there is remainder, the last store will be predicated. */
    int i;
    for (i = 0; i < cnt; i++)
    {
        V tmp = strm_eng<0, V>::get_adv();
        __vpred pred = strm_agen<0, V>::get_vpred();
        V *addr = strm_agen<0, V>::get_adv(out);
        __vstore_pred(pred, addr, tmp);
    }

    __SE0_CLOSE();
    __SA0_CLOSE();
}
```


Copyright © 2026 Texas Instruments Incorporated

Chapter 6

Run-Time Environment

This chapter describes the C7000 C/C++ run-time environment. To ensure successful execution of C/C++ programs, it is critical that all run-time code maintain this environment.

6.1 Memory 155
6.2 Object Representation 158
6.3 Register Conventions 165
6.4 Function Structure and Calling Conventions 167
6.5 Accessing Linker Symbols in C and C++ 169
6.6 Run-Time-Support Arithmetic Routines 169
6.7 System Initialization 171

Copyright © 2026 Texas Instruments Incorporated

6.1 Memory

The C7000 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 48-bit address space is available in target memory.

Note

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

The C7000 compiler requires that all code and data lie within 2 GB of virtual address space. Use a linker command file to place your code and data within this 2 GB virtual address region. The C7000 compiler produces code that uses position-independent addressing to obtain the addresses of functions (in some cases) and to access statically allocated data in memory. Instructions that use position-independent addressing have limited reach.

Information about linker command files can be found in Section 12.5.

6.1.1 Sections

The compiler produces relocatable blocks of code and data called *sections*, which are allocated in memory in a variety of ways to conform to various system configurations. For information about sections and allocating them, see the introductory object file information in Chapter 8.

There are two basic types of sections:

- **Uninitialized sections** reserve space in memory (typically RAM). A program can use this space at run time to create and store variables. The compiler creates the following uninitialized sections:
  - The **.bss section** reserves space for uninitialized global and static variables. These variables are allocated by the assembler.
  - The **.common section** reserves space for uninitialized global and static variables. These variables are allocated by the linker. Unused uninitialized variables are typically created as common symbols (unless you specify --common=off), so that they can be excluded from the resulting application.
  - The **.stack section** reserves memory for the system stack.
  - The **.sysmem section** reserves space for dynamic memory allocation. This space is used by dynamic memory allocation routines, such as malloc(), calloc(), realloc(), or new(). If a C/C++ program does not use these functions, the compiler does not create the .sysmem section.
- **Initialized sections** contain data or executable code. Initialized sections are usually read-only; exceptions are noted below. The C/C++ compiler creates the following initialized sections:
  - The **.args section** contains the command argument for a host-based loader. See the --arg_size option.
  - The **.binit section** contains boot time copy tables. For details on BINIT, see Section 12.8.4.2.
  - The **.cinit section** is created only if you are using the --rom_model option. It contains tables for explicitly initialized global and static variables.
  - The **.got section** contains the global offset table. This section is writable.
  - The **.init_array section** contains the table for calling global constructors.
  - The **.ovly section** contains copy tables for unions in which different sections have the same run address.
  - The **.data section** reserves space for non-const, initialized global and static variables. This section is writable.
  - The **.c7xabi.exidx section** contains the index table for exception handling. The **.c7xabi.extab section** contains stack unwinding instructions for exception handling. See the --exceptions option.

Copyright © 2026 Texas Instruments Incorporated

- The ***.name*.load section** contains the compressed image of section *name*.See Section 12.8 for information on copy tables.
- The **.const section** contains string literals, floating-point constants, and data defined with the C/C++ qualifier *const* (provided the constant is not also defined as *volatile* or one of the exceptions described in Section 5.5.2). String literals are placed in the .const:.string subsection to enable greater link-time placement control.
- The **.text section** contains all the executable code and compiler-generated constants. This section is usually read-only.

Note

Use Only Code in Program Memory

With the exception of code sections, the initialized and uninitialized sections cannot be allocated into internal program memory.

The assembler creates the default sections .text, , and .data. You can instruct the compiler to create additional sections by using the CODE_SECTION and DATA_SECTION pragmas (see Section 5.8.5 and Section 5.8.8).

6.1.2 C/C++ System Stack

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

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

The run-time stack grows from the high addresses to the low addresses. The compiler uses the D15 register to manage this stack. D15 is the *stack pointer* (SP), which points to the next unused location on the stack.

The linker sets the stack size, creates a global symbol, __TI_STACK_SIZE, and assigns it a value equal to the stack size in bytes. The default stack size is 0x2000 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 Section 12.4.

At system initialization, SP is set to the first 8-byte (64-bit) aligned address that is 16 bytes before the end (highest numerical address) of the .stack section. The SP is 8-byte aligned so that most 64-bit and smaller objects do not cross memory bank boundaries, which are 64-bits in size, and by convention, SP will always point to a free 16-byte location to optimize stack usage on function calls.

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 when the space required is greater than the 16 bytes reserved for each stack frame.

For more information about the stack and stack pointer, see Section 6.4.

Note

Stack Overflow: The compiler provides no means to check for stack overflow during compilation or at run time. A stack overflow disrupts the run-time environment, causing your program to fail. Be sure to allow enough space for the stack to grow. You can use the --entry_hook option to add code to the beginning of each function to check for stack overflow; see Section 3.13.

Copyright © 2026 Texas Instruments Incorporated

6.1.3 Dynamic Memory Allocation

The run-time-support library supplied with the C7000 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 section. You can set the size of the .sysmem section by using the --heap_size=*size* option with the linker command. The linker also creates a global symbol, __TI_SYSMEM_SIZE, and assigns it a value equal to the size of the heap in bytes. The default size is 1K bytes. For more information on the --heap_size option, see Section 12.4.

If you use any C I/O function, the RTS library allocates an I/O buffer for each file you access. This buffer will be a bit larger than BUFSIZ, which is defined in stdio.h and defaults to 256. Make sure you allocate a heap large enough for these buffers or use setvbuf to change the buffer to a statically-allocated buffer.

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

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

Use a pointer and call the malloc function:

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

When allocating from a heap, make sure the size of the heap is large enough for the allocation. This is particularly important when allocating variable-length arrays.

Copyright © 2026 Texas Instruments Incorporated

6.2 Object Representation

This section explains how various data objects are sized, aligned, and accessed.

6.2.1 Data Type Storage

For basic (scalar) types, the minimum alignment of an object is the size of its type. The minimum alignment for an object with an array type is that specified by the type of its elements.

For general information about data types, see Section 5.3. Table 6-1 lists register and memory storage for various data types:

Table 6-1. Data Representation in Registers and Memory

Data Type	Register Storage	Memory Storage
char	Bits 0-7 of register	8 bits aligned to 8-bit boundary
unsigned char	Bits 0-7 of register	8 bits aligned to 8-bit boundary
short	Bits 0-15 of register	16 bits aligned to 16-bit boundary
unsigned short	Bits 0-15 of register	16 bits aligned to 16-bit boundary
int	Bits 0-31 of register	32 bits aligned to 32-bit boundary
unsigned int	Bits 0-31 of register	32 bits aligned to 32-bit boundary
long	Entire scalar register or bits 0-63 of vector register	64 bits aligned to 64-bit boundary
unsigned long	Entire scalar register or bits 0-63 of vector register.	64 bits aligned to 64-bit boundary
enum (1)	Bits 0-31 of register or entire register	32 bits aligned to 32-bit boundary or 64 bits aligned to 64-bit boundary
float	Bits 0-31 of register	32 bits aligned to 32-bit boundary
double	Entire scalar register or bits 0-63 of vector register.	64 bits aligned to 64-bit boundary
long double	Entire scalar register or bits 0-63 of vector register.	64 bits aligned to 64-bit boundary
struct	Members are stored as their individual types require.	Storage is a multiple of the alignment to the boundary of largest member type; members are stored and aligned as their individual types require.
array	Members are stored as their individual types require.	Members are stored as their individual types require. All arrays inside a structure are aligned according to the type of each element in the array.
pointer to data member	Entire scalar register or bits 0-63 of vector register.	64 bits aligned to 64-bit boundary
pointer to member function	Components stored as their individual types require	64 bits aligned to 64-bit boundary
cchar(2)	Bits 0-15 of register	16 bits aligned to 8-bit boundary
cshort	Bits 0-31 of register	32 bits aligned to 16-bit boundary
cint	Entire scalar register or bits 0-63 of vector register	64 bits aligned to 32-bit boundary
cfloat	Entire scalar register or bits 0-63 of vector register	64 bits aligned to 32-bit boundary
clong	Bits 0-127 of vector register	128 bits aligned to 64-bit boundary
cdouble	Bits 0-127 of vector register	128 bits aligned to 64-bit boundary

(1) For details about the size of an enum type, see Section 5.3.1.
(2) Complex data types are described in Section 5.3.2.

For vector data types, the minimum alignment of an object is specified by the type of its elements.

Copyright © 2026 Texas Instruments Incorporated

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). By default, the `char` type is signed.

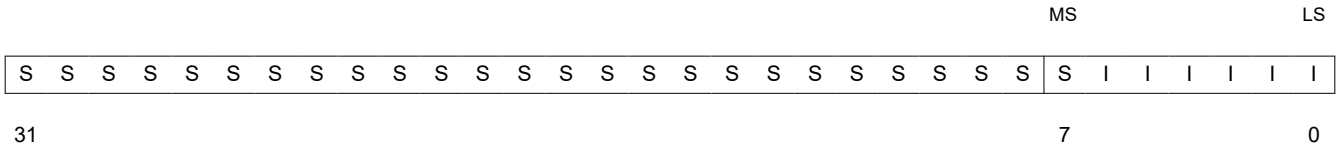
The `bool` type is also stored as an 8-bit type in bits 0-7 of a register.

Objects defined as `short` or `unsigned short` are stored in memory as two bytes at a halfword (2 byte) aligned address. They are loaded to and stored from bits 0-15 of a register (see Figure 6-1).

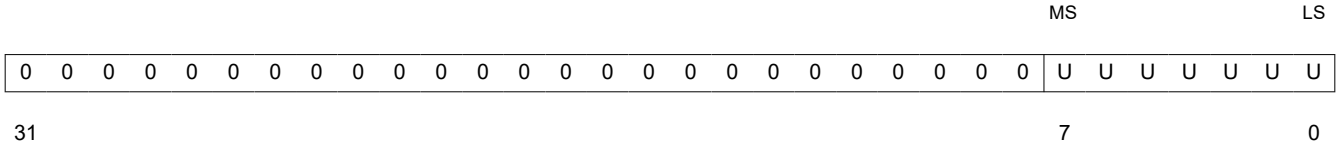
In big-endian mode, 2-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 8-15 of the register and moving the second byte of memory to bits 0-7. In little-endian mode, 2-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 0-7 of the register and moving the second byte of memory to bits 8-15.

Figure 6-1. Char and Short Data Storage Format

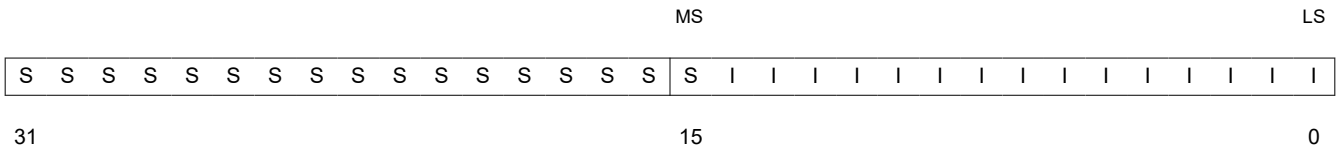
Signed 8-bit char



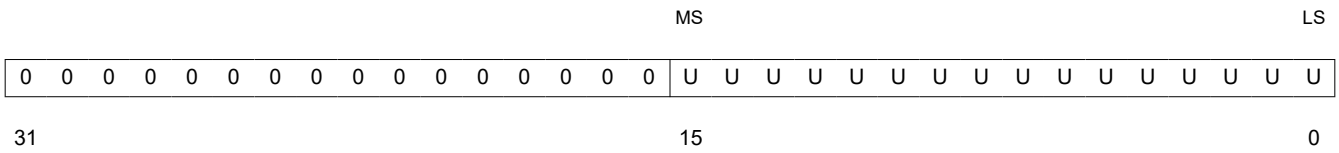
Unsigned 8-bit char



Signed 16-bit short



Unsigned 16-bit short



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

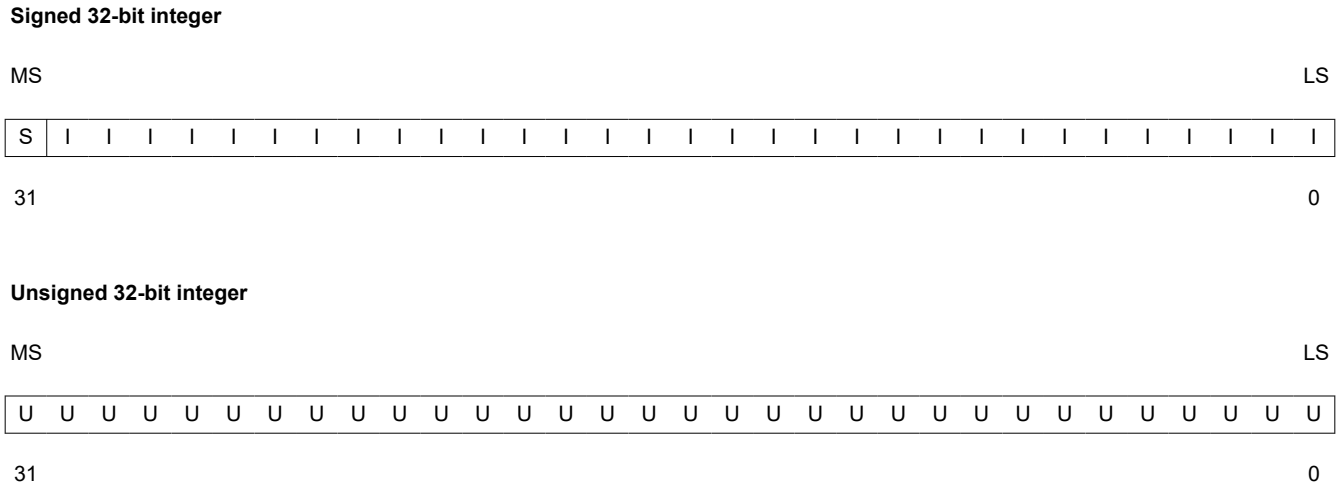
Copyright © 2026 Texas Instruments Incorporated

6.2.1.2 enum, int, and long Data Types (signed and unsigned)

The `int` and `unsigned int` data types are stored in memory as 32-bit objects (see Figure 6-2). Objects of these types are loaded to and stored from bits 0-31 of a register. In big-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 24-31 of the register, moving the second byte of memory to bits 16-23, moving the third byte to bits 8-15, and moving the fourth byte to bits 0-7. In little-endian mode, 4-byte objects are loaded to registers by moving the first byte (that is, the lower address) of memory to bits 0-7 of the register, moving the second byte to bits 8-15, moving the third byte to bits 16-23, and moving the fourth byte to bits 24-31.

For details about the size of an enum type, see Section 5.3.1.

Figure 6-2. 32-Bit Data Storage Format



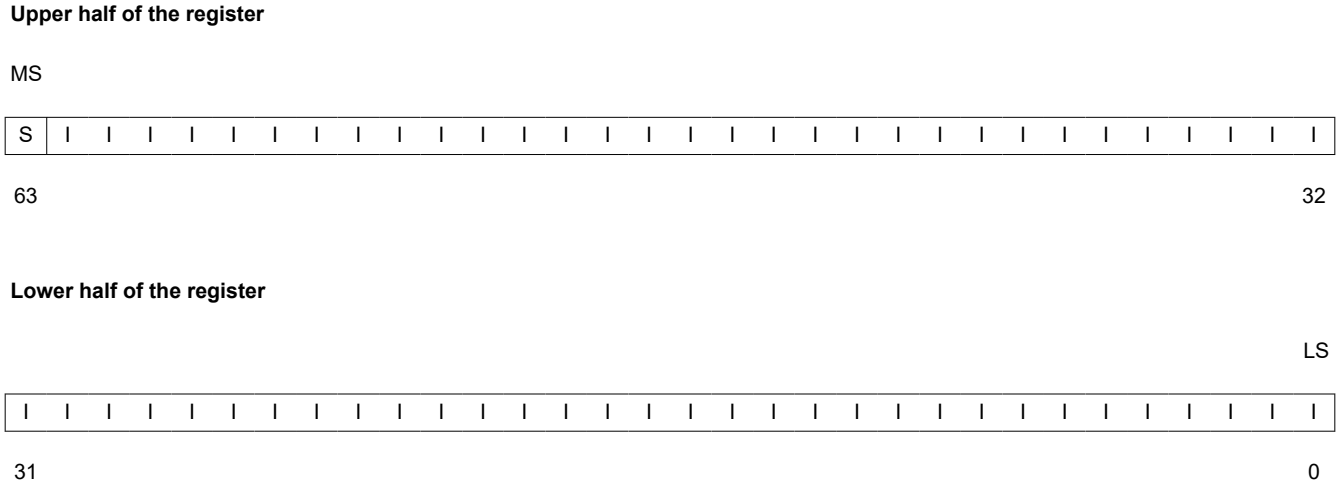
LEGEND: S = sign, U = unsigned integer, I = signed integer, MS = most significant, LS = least significant

Copyright © 2026 Texas Instruments Incorporated

6.2.1.3 long Data Types (signed and unsigned)

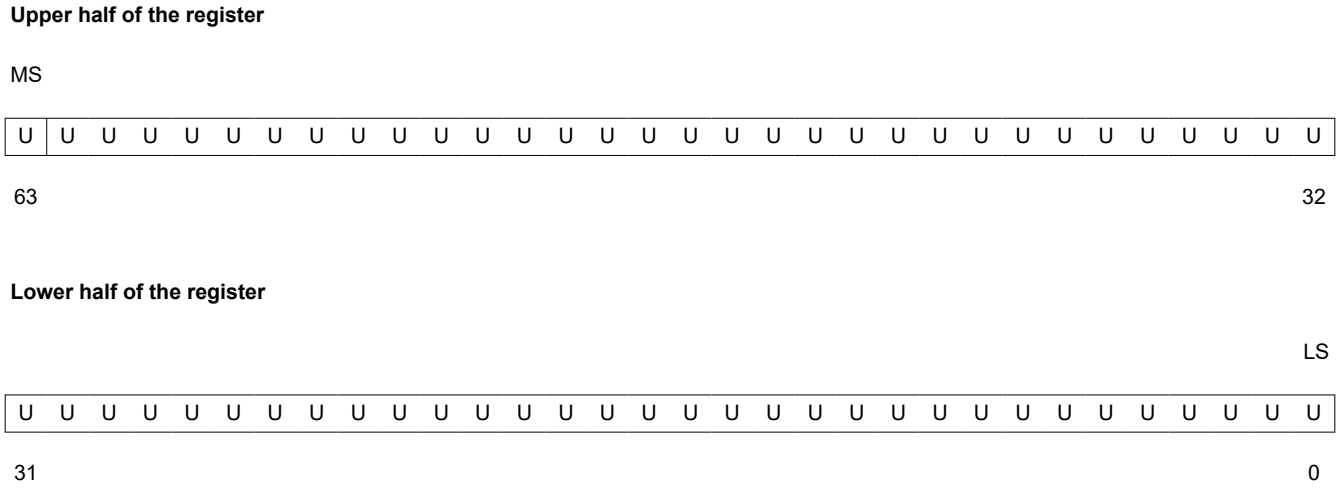
The `long` and `unsigned long` data types are stored in a full 64-bit register (see Figure 6-3). In big-endian mode, the lower address is loaded into bits 32-63 of the register and the higher address is loaded into bits 0-31 of the register. In little-endian mode, the lower address is loaded into bits 0-31 of the register and the higher address is loaded into bits 32-63s of the register.

Figure 6-3. 64-Bit Data Storage Format Signed 64-bit long



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

Figure 6-4. Unsigned 64-bit long



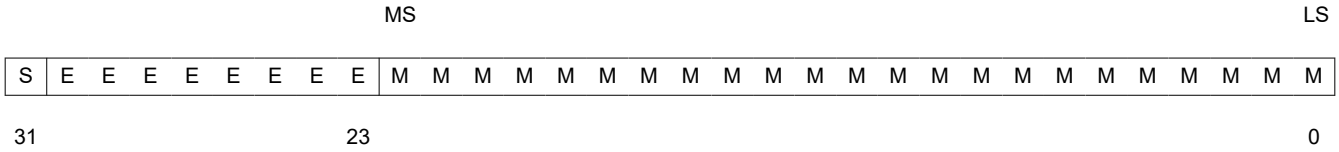
LEGEND: S = sign, U = unsigned integer, I = signed integer, X = unused, MS = most significant, LS = least significant

Copyright © 2026 Texas Instruments Incorporated

6.2.1.4 float Data Type

The float data type is stored in memory as 32-bit objects. The value is stored in the single-precision floating-point format (binary32) defined by the IEEE 754 standard.

Figure 6-5. Single-Precision Floating-Point Char Data Storage Format



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

6.2.1.5 double and long double Data Types

The `double` and `long double` data types are stored in a full 64-bit register. The value is stored in the double-precision floating-point format (binary64) defined by the IEEE 754 standard. That is, the exponent is 11 bits long, and the mantissa is 52 bits long.

6.2.1.6 Pointer to Data Member Types

Pointer to data member objects are stored in memory like an unsigned int (64-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.

6.2.1.7 Pointer to Member Function Types

Pointer to member function objects have a layout equivalent to:

```
struct __mptr {
   __vptp f;
   ptrdiff_t d;
};
```

where _vptp is the following union:

```
union {
   void (*f) ();
   int 0;
}
```

The parameter f is the pointer to the member function if it is nonvirtual. The 0 is the offset to the virtual function pointer within the class object. The parameter d is the offset to be added to the beginning of the class object for this pointer.

6.2.1.8 Structures and Arrays

A struct is aligned to a boundary required by the member it contains with the strictest alignment requirement. For example, if the largest alignment required by a member of the struct is 16-bit alignment (for example, a short), then the entire struct is aligned to a 16-bit boundary. If the struct contains a type that requires 64-bit alignment (such as a double or long), then the struct is aligned to a 64-bit boundary.

If a struct member is itself a struct, the size and alignment of the inner struct must be determined before the size and alignment of the outer struct may be determined.

Members of structs have sizes and alignments equal to those they would have as independent objects, unless the packed attribute is used. An array member of a struct is aligned to the alignment of its element type; this may differ from the alignment the element would have if it were an independent top-level (static) object.

Structs always have size equal to a multiple of the struct alignment. This sometimes requires padding after the last member to round the size up to a multiple of the struct alignment. The size of a structure includes any necessary padding between members. For example, if the largest member of a struct is of type float, the size of the struct will be a multiple of 32 bits.

Copyright © 2026 Texas Instruments Incorporated

Static scope arrays (sometimes called top-level arrays) are aligned on an 8-byte (64-bit) boundary.

6.2.2 Bit Fields

Bit fields are the only objects that are packed within a byte. That is, two bit fields can be stored in the same byte. Bit fields can range in size from 1 to 64 bits in C or larger in C++.

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

The size, alignment, and type of bit fields adhere to these rules:

- Bit fields up to long are supported.
- Bit fields are treated as the declared signed or unsigned type.
- The size and alignment of the struct containing a bit field depends on the declared type of the bit field. For example, consider the struct:

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

  This struct uses up 4 bytes and is aligned at 4 bytes.
- Unnamed bit fields do affect the alignment of the struct or union. For example, consider the struct:

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

  This struct uses 4 bytes and is aligned at a 4-byte boundary.
- Bit fields declared volatile are accessed according to the bit field's declared type. A volatile bit field reference generates exactly one reference to its storage; multiple volatile bit field accesses are not merged.

Figure 6-6 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.

Copyright © 2026 Texas Instruments Incorporated

Figure 6-6. Bit-Field Packing in Big-Endian and Little-Endian Formats

Big-endian register

MS LS

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

Little-endian register

MS LS

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

31 0

Little-endian memory

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

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

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 6.7.
- In an expression. For example:

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

  When a string is used in an expression, the string itself is defined in the .const:string section, along with a unique label that points to the string; the terminating 0 byte is explicitly added by the compiler.

  String labels have the form CSLn, where C is the compiler-generated symbol prefix and *n* is a number assigned by the compiler to make the label unique. The number begins at 0 and is increased by 1 for each string defined.

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

Copyright © 2026 Texas Instruments Incorporated

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

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

6.3 Register Conventions

Strict conventions associate specific registers with specific operations in the C/C++ environment.

The register conventions dictate how the compiler uses registers and how values are preserved across function calls.

The registers in Table 6-2 are available to the compiler for allocation to register variables and temporary expression results. If the compiler cannot allocate a register of a required type, spilling occurs. Spilling is the process of moving a register's contents to memory to free the register for another purpose.

The C7000 has two datapaths: an A-side datapath with 64-bit “scalar” registers, and a B-side datapath with “vector” registers, whose size depends on the C7000 target. (See Table 5-7 for target-specific vector widths.) The lower 64-bits of any B-side vector register can also be accessed as a scalar register by removing the ‘V’ from its name. Scalar registers are not limited to storing scalar values; a vector can be stored in a scalar register if it fits.

D15 is the Stack Pointer (SP). The stack pointer must always remain aligned on a 2-word (8-byte) boundary. The SP points at the first aligned address below (less than) the currently allocated stack.

RP, D15 (SP), A8-A15, B14/VB14, and B15/VB15 are *callee-save* registers. That is, a called function is required to preserve them so they have the same value on return from a function as they had at the point of the call.

All other registers are *caller-save*; that is, they are not preserved across a call, so if their value is needed following the call, the caller is responsible for saving and restoring their contents.

Table 6-2. Register Usage

Register	File	Preserved by Callee?	Role in Calling Convention
A0	A side scalar	no	
A1		no	Pointer to return-by-reference value
A2		no	
A3		no	
A4		no	1st scalar argument
A5		no	2nd scalar argument
A6		no	3rd scalar argument
A7		no	4th scalar argument
A8		yes	5th scalar argument
A9		yes	6th scalar argument
A10		yes	7th scalar argument
A11		yes	8th scalar argument
A12		yes	9th scalar argument
A13		yes	
A14		yes	
A15		yes	
AL0-AL7	A side local L	no	
AM0-AM7	A side local M	no	

Copyright © 2026 Texas Instruments Incorporated

Table 6-2. Register Usage (continued)

Register	File	Preserved by Callee?	Role in Calling Convention
VB0	B side vector	no	1st vector argument
VB1		no	2nd vector argument
VB2		no	3rd vector argument
VB3		no	4th vector argument
VB4		no	5th vector argument
VB5		no	6th vector argument
VB6		no	7th vector argument
VB7		no	8th vector argument
VB8		no	9th vector argument
VB9		no	10th vector argument
VB10		no	11th vector argument
VB11		no	12th vector argument
VB12		no	13th vector argument
VB13		no	14th vector argument
VB14		yes	15th vector argument
VB15		yes	16th vector argument
VBL0-VBL7	B side local L	no	
VBM-VBM7	B side local M	no	
D0-D14	D unit local	no	
D15		yes	Stack Pointer
RP	Control	yes	Return Pointer
P0	Vector predicates	no	1st vector predicate argument
P1		no	2nd vector predicate argument
P2		no	3rd vector predicate argument
P3		no	4th vector predicate argument
P4		no	5th vector predicate argument
P5		no	6th vector predicate argument
P6		no	7th vector predicate argument
P7		no	8th vector predicate argument
CUCR0-CUCR3	C-unit Control Register	no	

All other control registers are not saved or restored by the compiler.

The compiler assumes that control registers not listed in Table 6-2 that can have an effect on compiled code have default values.

Copyright © 2026 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.

6.4.1 How a Function Makes a Call

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

The C7000 has dedicated instructions and registers to manage call and return operations. The CALL instruction saves the return address in the Return Pointer (RP) register and transfers control to the called function. The RET instruction restores the PC from the RP, thereby returning control to the callee.

The C7000 CPU has a pipeline model that can function in both a protected or unprotected state. The CPU must be in the protected state. not unprotected state, when a call or return is made.

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

- Arguments passed to a function are placed in registers or on the stack.
  - If arguments are passed to a function, as many as can fit are placed in the nine "scalar" and sixteen "vector" registers that are available for argument passing.
  - Arguments whose declared type is 64 bits or less are assigned to scalar registers A4 through A12.
  - Arguments whose declared type is larger than 64 bits but less than or equal to the target vector width (see Table 5-7) are passed in vector registers VB0-VB15.
  - Arguments that are declared as vector predicates are passed in vector predicate registers P0-P7.
  - Arguments larger than the target vector width (256 bits, 512 bits or 1024 bits, depending on the target) are passed by reference.
  - Any remaining arguments are placed on the stack at increasing addresses, such that the first one will be at address SP+16 upon entry to the callee.
  - Each argument with a scalar or vector type is placed at the next available address correctly aligned for its type.
  - Structures are aligned to the next power of two greater than or equal to their size, up to a maximum of 8 bytes.
  - Each argument reserves an amount of stack space equal to its size rounded up to the next multiple of its alignment.
  - For a variadic C function (declared with an ellipsis indicating that it is called with varying numbers of arguments), the last explicitly declared argument and all remaining arguments are passed on the stack, so that its stack address can act as a reference for accessing the undeclared arguments.
  - An argument that is not declared in a prototype and whose size is less than the size of int is passed as an int, in accordance with the C language.
- The calling function must save register RP so that it is not overwritten by the CALL instruction. It must also save any live local or global registers that are not preserved by the called function (as part of the save-on-entry register set). The save-on-entry register set includes A8-A15, B14/VB14, and B15/VB15.
- The caller (parent) calls the function (child).

Copyright © 2026 Texas Instruments Incorporated

6.4.2 How a Called Function Responds

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

1. The called function (child) already has 16 bytes of reserved stack frame space available, but if the space required to store its local variables, temporary storage areas, and arguments to functions is larger than 16 bytes, then the called function allocates additional space on the stack.
2. If the called function calls any other functions, it must make sure that 16 bytes of unused space is also allocated on the stack and reserved for its called functions. The called function must also save its return address on the stack. Otherwise, it is left in the return register (RP) and is overwritten by the next function call.
3. If the called function modifies any save-on-entry registers (A8-A15, VB14-VB15), it must save them, either in other registers or on the stack. The called function can modify any other registers without saving them because they will have been saved by a caller function prior to the call if they were used.
4. If the called function expects a structure argument, effort is made to pass the structure by value if its value can fit into a scalar 64-bit register or a vector register. Otherwise, the called function receives a pointer to the structure instead. If writes are made to the structure from within the called function, space for a local copy of the structure must be allocated on the stack and the local structure must be copied from the passed pointer to the structure. If no writes are made to the structure, it can be referenced in the called function indirectly through the pointer argument.

   You must be careful to declare functions properly that accept structure arguments, both at the point where they are called (so that the structure argument is passed as an address) and at the point where they are declared (so the function knows to copy the structure to a local copy).
5. The called function executes the code for the function.
6. The return value is handled as follows:
   - If the called function returns any integer, pointer, float, double, long double, long type, or vector data type less than or equal to 64 bits in size, the return value is placed in register A4.
   - If the called function returns a vector data type greater than 64 bits in size, the return value is placed in register VB0.
   - If the called function returns a vector predicate type, the return value is placed in register P0.
   - If the called function returns a structure, it is returned by value if it is small enough to fit into register A4 or register VB0. Otherwise, the caller allocates space for the structure and passes the address of the return space to the called function in register A1. To return a structure, the called function copies the structure to the memory block pointer to by the extra argument.

   In this way, the caller can be smart about telling the called function where to return the structure. For example, in the statement s = f(x), where s is a structure and f is a function that returns a structure, the caller can actually make the call as f(&s, x). The function f then copies the return structure directly into s, performing the assignment automatically.

   If the caller does not use the return structure value, an address value of 0 can be passed as the first argument. This directs the called function not to copy the return structure.

   You must be careful to declare functions properly that return structures, both at the point where they are called (so that the extra argument is passed) and at the point where they are declared (so the function knows to copy the result).
7. Any save-on-entry register (A8-A15, B14/VB14, B15/VB15) that was saved in Step 3 is restored.
8. The value of the return register (RP) is also restored if it was saved.
9. Any space that was allocated during this call sequence is reclaimed.
10. The function returns by invoking the RET instruction, which returns to the location contained in the return register (RP).

Copyright © 2026 Texas Instruments Incorporated

6.4.3 Accessing Arguments and Local Variables

A function accesses its stack arguments and local nonregister variables indirectly through register D15 (SP), which points to the top of the stack. Since the stack grows toward smaller addresses, the local and argument data for a function are accessed with a positive offset from SP. Local variables, temporary storage, and the area reserved for stack arguments to functions called by this function are accessed with offsets from the SP.

For more information, see Section 6.4.2. For more information on the C/C++ System stack, see Section 6.1.2.

6.5 Accessing Linker Symbols in C and C++

See Section 12.6 for information about referring to linker symbols in C/C++ code.

6.6 Run-Time-Support Arithmetic Routines

The run-time-support library contains a number of assembly language functions that provide arithmetic routines for C/C++ math operations that the C7000 instruction set does not provide, such as integer division, integer remainder, and floating-point operations.

These routines follow the standard C/C++ calling sequence. The compiler automatically adds these routines when appropriate; they are not intended to be called directly by your programs.

The source code for these functions is provided in the `lib/src` source directory. The source code has comments that describe the operation of the functions. You can extract, inspect, and modify any of the math functions. Be sure, however, that you follow the calling conventions and register-saving rules outlined in this chapter. Table 6-3 summarizes the run-time-support functions used for arithmetic.

Copyright © 2026 Texas Instruments Incorporated

Table 6-3. C7000 Run-Time-Support Arithmetic Functions

Return Type	C Function	Description
void	__c7xabi_abort_msg (const char *)	Report failed assertion. (See notes below.)
double	__c7xabi_divd (double, double)	Divide two double-precision floats.
float	__c7xabi_divf (float, float)	Divide two single-precision floats.
long long	__c7xabi_divlli (long long, long long)	64-bit signed integer division.
unsigned long long	__c7xabi_divull (unsigned long long, unsigned long long)	64-bit unsigned integer division.
long long	__c7xabi_fixdlli (double)	Convert double-precision float to 64-bit integer.
unsigned	__c7xabi_fixdu (double)	Convert double-precision float to 32-bit unsigned integer.
unsigned long long	__c7xabi_fixdull (double)	Convert double-precision float to 64-bit unsigned integer.
long long	__c7xabi_fixflli (float)	Convert single-precision float to 64-bit integer.
unsigned	__c7xabi_fixfu (float)	Convert single-precision float to 32-bit unsigned integer.
unsigned long long	__c7xabi_fixfull (float)	Convert single-precision float to 64-bit unsigned integer.
double	__c7xabi_fltllid (long long)	Convert 64-bit integer to double-precision float.
float	__c7xabi_fltllif (long long)	Convert 64-bit integer to single-precision float.
double	__c7xabi_fltulld (unsigned long long)	Convert 64-bit unsigned integer to double-precision float.
float	__c7xabi_fltullf (unsigned long long)	Convert 64-bit unsigned integer to single-precision float.
long long	__c7xabi_remlli (long long, long long)	64-bit integer modulo.
unsigned long long	__c7xabi_remull (unsigned long long, unsigned long long)	64-bit unsigned integer modulo.
void	__c7xabi_strasg (int*, const int*, unsigned)	Block copy. (See notes below.)
	__c7xabi_unwind_cpp_pr0	Short frame unwinding, 16-bit scope.
	__c7xabi_unwind_cpp_pr1	Long frame unwinding, 16-bit scope.
	__c7xabi_unwind_cpp_pr2	Long frame unwinding, 32-bit scope.
	__c7xabi_unwind_cpp_pr3	Unwinding, 24-bit encoding, 16-bit scope.

__c7xabi_abort_msg() Function:

```
void __c7xabi_abort_msg(const char *msg)
```

The function `__c7xabi_abort_msg()` is generated to print a diagnostic message when a run-time assertion (for example, the C assert macro) fails. It must not return. That is, it must call abort or terminate the program by other means.

__c7xabi_strasg() Function:

```
void __c7xabi_strasg(int* dst, const int* src, unsigned  cnt)
```

The function `__c7xabi_strasg()` is generated by the compiler for efficient out-of-line structure or array copy operations. The `cnt` argument is the size in bytes.

Copyright © 2026 Texas Instruments Incorporated

6.7 System Initialization

Before you can run a C/C++ program, you must create the C/C++ run-time environment. The C/C++ boot routine performs this task using a function called c_int00 (or _c_int00). The run-time-support source library, rts.src, contains the source to this routine in a module named boot.c (or boot.asm).

To begin running the system, the c_int00 function can be branched to or called, but it is usually vectored to by reset hardware. You must link the c_int00 function with the other object files. This occurs automatically when you use the --rom_model or --ram_model link option and include a standard run-time-support library as one of the linker input files.

When C/C++ programs are linked, the linker sets the entry point value in the executable output file to the symbol c_int00. The c_int00 function performs the following tasks to initialize the environment:

1. Defines a section called .stack for the system stack and sets up the initial stack pointers
2. Performs C autoinitialization of global/static variables. For more information, see Section 6.7.2.
3. Initializes global variables by copying the data from the initialization tables to the storage allocated for the variables in the .bss section. If you are initializing variables at load time (--ram_model option), a loader performs this step before the program runs (it is not performed by the boot routine).
4. Calls C++ initialization routines for file scope construction from the global constructor table. For more information, see Section 6.7.2.6.
5. Calls the main() function to run the C/C++ program

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the operations listed above to correctly initialize the C/C++ environment.

6.7.1 Boot Hook Functions for System Pre-Initialization

Boot hooks are points at which you may insert application functions into the C/C++ boot process. Default boot hook functions are provided with the run-time support (RTS) library. However, you can implement customized versions of these boot hook functions, which override the default boot hook functions in the RTS library if they are linked before the run-time library. Such functions can perform any application-specific initialization before continuing with the C/C++ environment setup.

The following boot hook functions are available:

_system_pre_init(): This function provides a place to perform application-specific initialization. It is invoked after the stack pointer is initialized but before any C/C++ environment setup is performed. By default, _system_pre_init() should return a non-zero value. The default C/C++ environment setup is bypassed if _system_pre_init() returns 0.

_system_post_cinit(): This function is invoked during C/C++ environment setup, after C/C++ global data is initialized but before any C++ constructors are called. This function should not return a value.

6.7.2 Automatic Initialization of Variables

Any global variables declared as preinitialized must have initial values assigned to them before a C/C++ program starts running. The process of retrieving these variables' data and initializing the variables with the data is called autoinitialization. Internally, the compiler and linker coordinate to produce compressed initialization tables. Your code should not access the initialization table.

6.7.2.1 Zero Initializing Variables

In ANSI C, global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ compiler supports preinitialization of uninitialized variables by default. This can be turned off by specifying the linker option --zero_init=off.

Zero initialization takes place only if the --rom_model linker option, which causes autoinitialization to occur, is used. If you use the --ram_model option for linking, the linker does not generate initialization records, and the loader must handle both data and zero initialization.

Copyright © 2026 Texas Instruments Incorporated

6.7.2.2 Direct Initialization

The compiler uses direct initialization to initialize global variables. For example, consider the following C code:

```
int i    = 23;
int a[5] = { 1, 2, 3, 4, 5 };
```

The compiler allocates the variables 'i' and 'a[] to .data section and the initial values are placed directly.

```
        .global i
        .data
        .align  4
i:
        .field          23,32                   ; i @ 0
        .global a
        .data
        .align  4
a:
        .field          1,32                    ; a[0] @ 0
        .field          2,32                    ; a[1] @ 32
        .field          3,32                    ; a[2] @ 64
        .field          4,32                    ; a[3] @ 96
        .field          5,32                    ; a[4] @ 128
```

Each compiled module that defines static or global variables contains these .data sections. The linker treats the .data section like any other initialized section and creates an output section. In the load-time initialization model, the sections are loaded into memory and used by the program. See Section 6.7.2.5.

In the run-time initialization model, the linker uses the data in these sections to create initialization data and an additional compressed initialization table. The boot routine processes the initialization table to copy data from load addresses to run addresses. See Section 6.7.2.3.

6.7.2.3 Autoinitialization of Variables at Run Time

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the --rom_model option.

Using this method, the linker creates a compressed initialization table and initialization data from the direct initialized sections in the compiled module. The table and data are used by the C/C++ boot routine to initialize variables in RAM using the table and data in ROM.

Figure 6-7 illustrates autoinitialization at run time. Use this method in any system where your application runs from code burned into ROM.

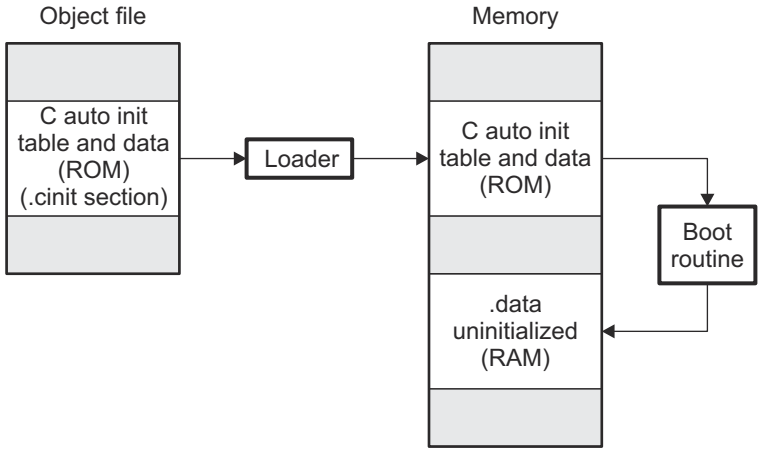


Figure 6-7. Autoinitialization at Run Time

Copyright © 2026 Texas Instruments Incorporated

6.7.2.4 Autoinitialization Tables

The compiled object files do not have initialization tables. The variables are initialized directly. The linker, when the --rom_model option is specified, creates C auto initialization table and the initialization data. The linker creates both the table and the initialization data in an output section named .cinit.

The autoinitialization table has the following format:

_TI_CINIT_Base:

48-bit load address	48-bit run address
⋮	⋮
48-bit load address	48-bit run address

_TI_CINIT_Limit:

The linker defined symbols __TI_CINIT_Base and __TI_CINIT_Limit point to the start and end of the table, respectively. Each entry in this table corresponds to one output section that needs to be initialized. The initialization data for each output section could be encoded using different encoding.

The load address in the C auto initialization record points to initialization data with the following format:

8-bit index	Encoded data

The first 8-bits of the initialization data is the handler index. It indexes into a handler table to get the address of a handler function that knows how to decode the following data.

The handler table is a list of 32-bit function pointers.

_TI_Handler_Table_Base:

48-bit handler 1 address
⋮
48-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.

6.7.2.4.1 Length Followed by Data Format

8-bit index	24-bit padding	32-bit length (N)	N byte initialization data (not compressed)

The compiler uses 24-bit padding to align the length field to a 32-bit boundary. The 32-bit length field encodes the length of the initialization data in bytes (N). N byte initialization data is not compressed and is copied to the run address as is.

The run-time support library has a function __TI_zero_init() to process this type of initialization data. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

Copyright © 2026 Texas Instruments Incorporated

6.7.2.4.2 Zero Initialization Format

8-bit index	24-bit padding	32-bit length (N)

The compiler uses 24-bit padding to align the length field to a 32-bit boundary. The 32-bit length field encodes the number of bytes to be zero initialized.

The run-time support library has a function __TI_zero_init() to process the zero initialization. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

6.7.2.4.3 Run Length Encoded (RLE) Format

8-bit index	Initialization data compressed using run length encoding

The data following the 8-bit index is compressed using Run Length Encoded (RLE) format. uses a simple run length encoding that can be decompressed using the following algorithm:

1. Read the first byte, Delimiter (D).
2. Read the next byte (B).
3. If B != D, copy B to the output buffer and go to step 2.
4. Read the next byte (L).
   a. If L == 0, then length is either a 16-bit, a 24-bit value, or we've reached the end of the data, read next byte (L).
      i. If L == 0, length is a 24-bit value or the end of the data is reached, read next byte (L).
         1. If L == 0, the end of the data is reached, go to step 7.
         2. Else L <<= 16, read next two bytes into lower 16 bits of L to complete 24-bit value for L.
      ii. Else L <<= 8, read next byte into lower 8 bits of L to complete 16-bit value for L.
   b. Else if L > 0 and L < 4, copy D to the output buffer L times. Go to step 2.
   c. Else, length is 8-bit value (L).
5. Read the next byte (C); C is the repeat character.
6. Write C to the output buffer L times; go to step 2.
7. End of processing.

The run-time support library has a routine __TI_decompress_rle24() to decompress data compressed using RLE. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

Note

RLE Decompression Routine: The previous decompression routine, __TI_decompress_rle(), is included in the run-time-support library for decompressing RLE encodings generated by older versions of the linker.

6.7.2.4.4 Lempel-Ziv-Storer-Szymanski Compression (LZSS) Format

8-bit index	Initialization data compressed using LZSS

The data following the 8-bit index is compressed using LZSS compression. The run-time support library has the routine __TI_decompress_lzss() to decompress the data compressed using LZSS. The first argument to this function is the address pointing to the byte after the 8-bit index. The second argument is the run address from the C auto initialization record.

Copyright © 2026 Texas Instruments Incorporated

6.7.2.5 Initialization of Variables at Load Time

Initialization of variables at load time enhances performance by reducing boot time and by saving the memory used by the initialization tables. To use this method, invoke the linker with the --ram_model option.

When you use the --ram_model link option, the linker does not generate C autoinitialization tables and data. The direct initialized sections (.data) in the compiled object files are combined according to the linker command file to generate initialized output sections. The loader loads the initialized output sections into memory. After the load, the variables are assigned their initial values.

Since the linker does not generate the C autoinitialization tables, no boot time initialization is performed.

Figure 6-8 illustrates the initialization of variables at load time.

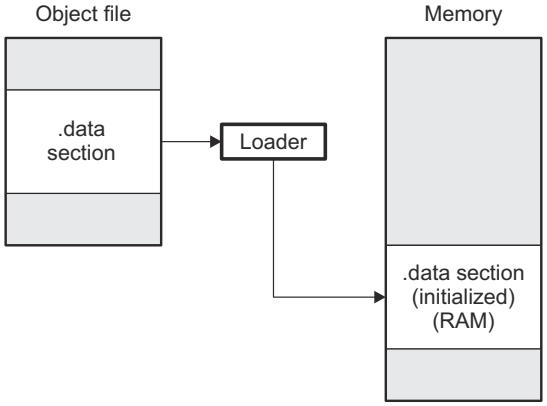


Figure 6-8. Initialization at Load Time

6.7.2.6 Global Constructors

All global C++ variables that have constructors must have their constructor called before main(). The compiler builds a table of global constructor addresses that must be called, in order, before main() in a section called .init_array. The linker combines the .init_array section form each input file to form a single table in the .init_array section. The boot routine uses this table to execute the constructors. The linker defines two symbols to identify the combined .init_array table as shown below. This table is not null terminated by the linker.

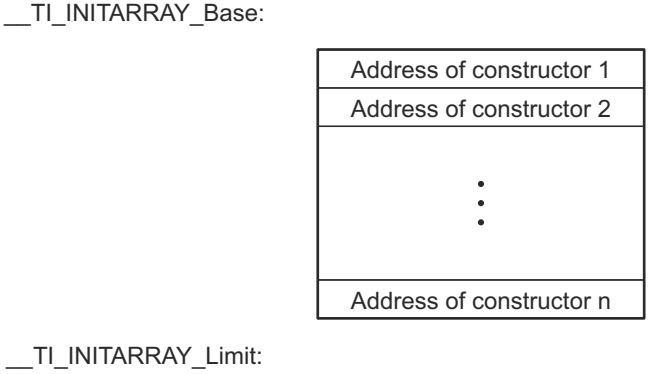


Figure 6-9. Constructor Table

Copyright © 2026 Texas Instruments Incorporated

Chapter 7

Using Run-Time-Support Functions and Building Libraries

Some of the features of C/C++ (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are provided as an ANSI/ISO C/C++ standard library, rather than as part of the compiler itself. The TI implementation of this library is the run-time-support library (RTS). The C/C++ compiler implements the ISO standard library except for those facilities that handle signal and locale issues (properties that depend on local language, nationality, or culture). Using the ANSI/ISO standard library ensures a consistent set of functions that provide for greater portability.

In addition to the ANSI/ISO-specified functions, the run-time-support library includes routines that give you processor-specific commands and direct C language I/O requests. These are detailed in Section 7.1 and Section 7.2.

A library-build utility is provided with the code generation tools that lets you create customized run-time-support libraries. This process is described in Section 7.4.

7.1 C and C++ Run-Time Support Libraries 177
7.2 The C I/O Functions 181
7.3 Handling Reentrancy (_register_lock() and _register_unlock() Functions) 195
7.4 Library-Build Process 196

Copyright © 2026 Texas Instruments Incorporated

7.1 C and C++ Run-Time Support Libraries

C7000 compiler releases include pre-built run-time support (RTS) libraries that provide all the standard capabilities. Separate libraries are provided for each target CPU version, big and little endian support, and C++ exception support. See Section 7.1.8 for information on the library-naming conventions.

The run-time-support library contains the following:

- ANSI/ISO C/C++ standard library
- C I/O library
- Low-level support functions that provide I/O to the host operating system
- Fundamental arithmetic routines
- System startup routine, _c_int00
- Compiler helper functions (to support language features that are not directly efficiently expressible in C/C++)

The run-time-support libraries do not contain functions involving signals and locale issues.

The C++ library supports wide chars, in that template functions and classes that are defined for char are also available for wide char. For example, wide char stream classes wios, wiostream, wstreambuf and so on (corresponding to char classes ios, iostream, streambuf) are implemented. However, there is no low-level file I/O for wide chars. Also, the C library interface to wide char support (through the C++ headers <cwchar> and <cwctype>) is limited as described in Section 5.1.

TI does not provide documentation that covers the functionality of the C++ library. TI suggests referring to one of the following sources:

- *The Standard C++ Library: A Tutorial and Reference*, Nicolai M. Josuttis, Addison-Wesley, ISBN 0-201-37926-0
- *The C++ Programming Language* (Third or Special Editions), Bjarne Stroustrup, Addison-Wesley, ISBN 0-201-88954-4 or 0-201-70073-5

7.1.1 Linking Code With the Object Library

When you link your program, you must specify the object library as one of the linker input files so that references to the I/O and run-time-support functions can be resolved. You can either specify the library or allow the compiler to select one for you. See Section 11.3.1 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 Chapter 12.

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 C7X_C_DIR environment variable to the include directory where the tools are installed.

The following header files provide TI extensions to the C standard:

- c7x.h -- Provides intrinsic definitions.
- c7x_direct.h -- Provides low-level "direct-mapped" intrinsic definitions. Automatically included by c7x.h.
- c7x_vpred.h -- Provides low-level vector predication intrinsic definitions. Automatically included by c7x.h.
- c7x_mma.h -- Provides MMA intrinsic definitions. Automatically included by c7x.h.
- c6x_migration.h -- Provides definitions of legacy C6000 intrinsics.
- c7x_iluthist.h -- Defines intrinsics for internal lookup table and histogram features. Automatically included by c7x.h.
- c7x_luthist.h -- Defines intrinsics for lookup table and histogram features. Automatically included by c7x.h.
- c7x_scalable.h -- Utilities for vector-width independent code. Not included automatically. See Section 5.16.
- c7x_strm.h -- Defines intrinsics for the Streaming Engine (SE) and Streaming Address Generator (SA). (See Section 4.15.) Automatically included by c7x.h.

Copyright © 2026 Texas Instruments Incorporated

- cpy_tbl.h -- Declares the copy_in() RTS function, which is used to move code or data from a load location to a separate run location at run-time. This function helps manage overlays.
- _data_synch.h -- Declares functions used by the RTS library to help with shared data synchronization. For example, these functions are used when flushing the local data cache to global shared memory.
- file.h -- Declares functions used by low-level I/O functions in the RTS library.
- gsm.h -- Provides basic DSP operations and GSM math operations defined by the European Telecommunications Standards Institute (ETSI).
- _lock.h -- Used when declaring system-wide mutex locks. This header file is deprecated; use _reg_mutex_api.h and _mutex.h instead.
- memory.h -- Provides the memalign() function, which is not required by the C standard.
- _mutex.h -- Declares functions used by the RTS library to help facilitate mutexes for specific resources that are owned by the RTS. For example, these functions are used for heap or file table allocation.
- _pthread.h -- Declares low-level mutex infrastructure functions and provides support for recursive mutexes.
- _reg_mutex_api.h -- Declares a function that can be used by an RTOS to register an underlying lock mechanism and/or thread ID mechanism that is implemented in the RTOS but is called indirectly by the RTS' _mutex.h functions.
- _reg_synch_api.h -- Declares a function that can be used by an RTOS to register an underlying cache synchronization mechanism that is implemented in the RTOS but is called indirectly by the RTS' _data_synch.h functions.
- strings.h -- Provides additional string functions, including bcmp(), bcopy(), bzero(), ffs(), index(), rindex(), strcasecmp(), and strncasecmp(). See the v7.6 release notes for details on these functions.

The following standard C header files are provided with the compiler: assert.h, complex.h, ctype.h, errno.h, float.h, inttypes.h, iso646.h, limits.h, locale.h, math.h, setjmp.h, signal.h, stdarg.h, stdbool.h, stddef.h, stdint.h, stdio.h, stdlib.h, string.h, time.h, wchar.h, and wctype.h.

The following standard C++ header files are provided with the compiler: algorithm, bitset, cassert, cctype, cerrno, cfloat, ciso646, climits, clocale, cmath, complex, csetjmp, csignal, cstdarg, cstddef, cstdio, cstdlib, cstring, ctime, cwchar, cwctype, deque, exception, fstream, functional, hash_map, hash_set, iomanip, ios, iosfwd, iostream, istream, iterator, limits, list, locale, map, memory, new, numeric, ostream, queue, rope, set, sstream, stack, stdexcept, streambuf, string, strstream, typeinfo, utility, valarray, and vector.

The following header files are for use with older C++ code: fstream.h, iomanip.h, iostream.h, new.h, stdiostream.h, stl.h, and strstream.h.

The following header files are for internal use by TI components and should not be directly included by your applications: _data_synch.h, _fmt_specifier.h, _isfuncdcl.h, _isfuncdef.h, _mutex.h, _pthread.h, access.h, c60asm.i, cpp_inline_math.h, elf_linkage.h, elfnames.h, linkage.h, mathf.h, mathl.h, pprof.h, unaccess.h, wchar.hx, xcomplex, xdebug, xhash, xiosbase, xlocale, xlocinfo, xlocinfo.h, xlocmes, xlocmon, xlocnum, xloctime, xmemory, xstddef, xstring, xtree, xutility, xwcc.h, ymath.h, and yvals.h,

7.1.3 Modifying a Library Function

You can inspect or modify library functions by examining the source code in the lib/src subdirectory of the compiler installation. For example, `C:\ti\ccsv7\tools\compiler\c7000_#.#.#\lib\src`.

One you have located the relevant source code, change the specific function file and rebuild the library. You can use this source tree to rebuild the rts7100_le.lib, rts7100_le_eh.lib, rts7100_be.lib, or rts7100_be_eh.lib library or to build a new library. See Section 7.1.8 for details on library naming and Section 7.4 for details on building

Copyright © 2026 Texas Instruments Incorporated

7.1.4 Support for String Handling

The RTS library provides the standard C header file <string.h> as well as the POSIX header file <strings.h>, which provides additional functions not required by the C standard. The POSIX header file <strings.h> provides:

- bcmp(), which is equivalent to memcmp()
- bcopy(), which is equivalent to memmove()
- bzero(), which is equivalent to memset(.., 0, ...);
- ffs(), which finds the first bit set and returns the index of that bit
- index(), which is equivalent to strchr()
- rindex(), which is equivalent to strrchr()
- strcasecmp() and strncasecmp(), which perform case-insensitive string comparisons

In addition, the header file <string.h> provides one additional function not required by the C standard.

- strdup(), which duplicates a string by dynamically allocating memory (as if by using malloc) and copying the string to this allocated memory

7.1.5 Minimal Support for Internationalization

The library includes the header files <locale.h>, <wchar.h>, and <wctype.h>, which provide APIs to support non-ASCII character sets and conventions. Our implementation of these APIs is limited in the following ways:

- The library has minimal support for wide and multibyte characters. The type wchar_t is implemented as unsigned int. The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h> but does not include all the functions specified in the standard. See Section 5.4 for more information about extended character sets.
- The C library includes the header file <locale.h> but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hard-coded to the behavior of the C locale, and attempting to install a different locale via a call to setlocale() will return NULL.

7.1.6 Support for Time and Clock Functions

The compiler RTS library supports two low-level time-related standard C functions in `time.h`:

- `clock_t clock(void);`

  The clock() function returns the number of clock cycles since the program began executing; the clock() function has nothing to do with wall-clock time.
- `time_t time(time_t *timer);`

  The time() function returns the wall-clock time.

Time-related and clock-related RTS library source files are provided in the `/lib/src` directory of the compiler installation.

The default implementations of these functions require that the program be run under CCS or a similar tool that supports the CIO System Call Protocol. If CIO is not available and you need to use one of these functions, you must provide your own definition of the function.

The clock() function returns the number of clock cycles since the program began executing. This information is available in a register on some devices, but the location varies from platform to platform. The compiler's RTS library provides an implementation that uses the CIO System Call Protocol to communicate with CCS, which figures out how to compute the right value for this device.

If CCS is not available, you must provide an implementation of the clock() function that gathers clock cycle information from the appropriate location on the device.

The time() function returns the real-world time, in terms of seconds since an epoch.

On many embedded systems, there is no internal real-world clock, so a program needs to discover the time from an external source. The compiler's RTS library provides an implementation that uses the CIO System Call Protocol to communicate with CCS, which provides the real-world time.

Copyright © 2026 Texas Instruments Incorporated

If CCS is not available, you must provide an implementation of the time() function that finds the time from some other source. If the program is running under an operating system, that operating system typically provides an implementation of time().

The time() function returns the number of seconds since an *epoch*. On C7000 systems, the epoch is defined as the number of seconds since midnight UTC January 1, 1970.

The TI RTS library does not provide time zone lookup tables. The local time zone is specified by a global variable `_tz` of type struct TZ, which is defined in `tmzone.c` in the RTS library source code. This struct is used by the mktime() and strftime() functions. To change the local time zone, assign new values to the following fields in this struct:

- Set the `_tz.daylight` field to the exact value for the mktime() function to assign to the struct `tm` member `tm_isdst`.
- Set the `_tz.timezone` field to a signed integer representing the number of seconds west of UTC. For example, Central Standard Time (UTC+6) is 21600 seconds west of UTC. Use a negative value for timezones east of UTC.
- Set the `_tz.tzname` field to the abbreviated name of the local time zone. This string is used verbatim by the strftime() function for the %Z conversion. For example, if the local time zone is Central Standard Time, use "CST" for this field.
- The `_tz.dstname` field is currently unused by the RTS library.

7.1.7 Allowable Number of Open Files

In the <stdio.h> header file, the value for the macro FOPEN_MAX has the value of the macro _NFILE, which is set to 10. The impact is that you can only have 10 files simultaneously open at one time (including the pre-defined streams - stdin, stdout, stderr).

The C standard requires that the minimum value for the FOPEN_MAX macro is 8. The macro determines the maximum number of files that can be opened at one time. The macro is defined in the stdio.h header file and can be modified by changing the value of the _NFILE macro and recompiling the library.

7.1.8 Library Naming Conventions

By default, the linker uses automatic library selection to select the correct run-time-support library (see Section 11.3.1.1) for your application. If you select the library manually, you must select the matching library using a naming scheme like the following:

rts7100_[*endian*][_eh].lib

endian	Indicates endianness:	
	le	Little-endian library
	be	Big-endian library
eh	Indicates whether the library has exception handling support (_eh) or not (blank).	

Copyright © 2026 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.

Note

Debugger Required for Default HOST: For the default HOST device to work, there must be a debugger to handle the C I/O requests; the default HOST device cannot work by itself in an embedded system. To work in an embedded system, you will need to provide an appropriate driver for your system.

Note

C I/O Mysteriously Fails: If there is not enough space on the heap for a C I/O buffer, operations on the file will silently fail. If a call to printf() mysteriously fails, this may be the reason. The heap needs to be at least large enough to allocate a block of size BUFSIZ (defined in stdio.h) for every file on which I/O is performed, including stdout, stdin, and stderr, plus allocations performed by the user's code, plus allocation bookkeeping overhead. Alternately, declare a char array of size BUFSIZ and pass it to setvbuf to avoid dynamic allocation. To set the heap size, use the --heap_size option when linking Section 12.4.

Note

Open Mysteriously Fails: The run-time support limits the total number of open files to a small number relative to general-purpose processors. If you attempt to open more files than the maximum, you may find that the open will mysteriously fail. You can increase the number of open files by extracting the source code from rts.src and editing the constants controlling the size of some of the C I/O data structures. The macro _NFILE controls how many FILE (fopen) objects can be open at one time (stdin, stdout, and stderr count against this total). (See also FOPEN_MAX.) The macro _NSTREAM controls how many low-level file descriptors can be open at one time (the low-level files underlying stdin, stdout, and stderr count against this total). The macro _NDEVICE controls how many device drivers are installed at one time (the HOST device counts against this total).

7.2.1 High-Level I/O Functions

The high-level functions are the standard C library of stream I/O routines (printf, scanf, fopen, getchar, and so on). These functions call one or more low-level I/O functions to carry out the high-level I/O request. The high-level I/O routines operate on FILE pointers, also called *streams*.

Portable applications should use only the high-level I/O functions.

To use the high-level I/O functions, include the header file stdio.h, or cstdio for C++ code, for each module that references a C I/O function. For example, given the following C program in a file named main.c:

```
#include <stdio.h>
void main()
{
   FILE *fid;
   fid = fopen("myfile","w");
   fprintf(fid,"Hello, world\n");
   fclose(fid);
   printf("Hello again, world\n");
}
```


Copyright © 2026 Texas Instruments Incorporated

Issuing the following compiler command compiles, links, and creates main.out from the run-time-support library:

```
cl7x main.c -z --heap_size=1000 --output_file=main.out
```

Executing main.out results in

```
Hello, world
```

being output to a file and

```
Hello again, world
```

being output to your host's stdout window.

7.2.1.1 Formatting and the Format Conversion Buffer

The internal routine behind the C I/O functions—such as printf(), vsnprintf(), and snprintf()—reserves stack space for a format conversion buffer. The buffer size is set by the macro FORMAT_CONVERSION_BUFFER, which is defined in format.h. Consider the following issues before reducing the size of this buffer:

- The default buffer size is 510 bytes. If MINIMAL is defined, the size is set to 32, which allows integer values without width specifiers to be printed.
- Each conversion specified with %xxxx (except %s) must fit in FORMAT_CONVERSION_BUFSIZE. This means any individual formatted float or integer value, accounting for width and precision specifiers, needs to fit in the buffer. Since the actual value of any representable number should easily fit, the main concern is ensuring the width and/or precision size meets the constraints.
- The length of converted strings using %s are unaffected by any change in FORMAT_CONVERSION_BUFSIZE. For example, you can specify `printf("%s value is %d", some_really_long_string, intval)` without a problem.
- The constraint is for each individual item being converted. For example a format string of `%d item1 %f item2 %e item3` does not need to fit in the buffer. Instead, each converted item specified with a % format must fit.
- There is no buffer overrun check.

7.2.2 Overview of Low-Level I/O Implementation

The low-level functions are comprised of seven basic I/O functions: open, read, write, close, lseek, rename, and unlink. These low-level routines provide the interface between the high-level functions and the device-level drivers that actually perform the I/O command on the specified device.

The low-level functions are designed to be appropriate for all I/O methods, even those which are not actually disk files. Abstractly, all I/O channels can be treated as files, although some operations (such as lseek) may not be appropriate. See Section 7.2.3 for more details.

The low-level functions are inspired by, but not identical to, the POSIX functions of the same names.

The low-level functions operate on file descriptors. A file descriptor is an integer returned by open, representing an opened file. Multiple file descriptors may be associated with a file; each has its own independent file position indicator.

Copyright © 2026 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 © 2026 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

Copyright © 2026 Texas Instruments Incorporated

write

Write Characters to a File

Syntax

#include <file.h>

int write (int *file_descriptor* **, const char** **buffer* **, unsigned***count* **);**

Description

The write function writes the number of characters specified by *count* from the *buffer* to the file associated with *file_descriptor*.

- The *file_descriptor* is the number assigned by open to an opened file.
- The *buffer* is where the characters to be written are located.
- The *count* is the number of characters to write to the file.

Return Value

The function returns one of the following values:

#	number of characters written if successful (may be less than *count*)
-1	on failure

lseek

Set File Position Indicator

Syntax for C

#include <file.h>

off_t lseek (int *file_descriptor* **, off_t***offset* **, int***origin* **);**

Description

The lseek function sets the file position indicator for the given file to a location relative to the specified origin. The file position indicator measures the position in characters from the beginning of the file.

- The *file_descriptor* is the number assigned by open to an opened file.
- The *offset* indicates the relative offset from the *origin* in characters.
- The *origin* is used to indicate which of the base locations the *offset* is measured from. The *origin* must be one of the following macros:

  SEEK_SET (0x0000) Beginning of file

  SEEK_CUR (0x0001) Current value of the file position indicator

  SEEK_END (0x0002) End of file

Return Value

The return value is one of the following:

#	new value of the file position indicator if successful
(off_t)-1	on failure

Copyright © 2026 Texas Instruments Incorporated

unlink

Delete File

Syntax

#include <file.h>

int unlink (const char *_path_ **);**

Description

The unlink function deletes the file specified by *path*. Depending on the device, a deleted file may still remain until all file descriptors which have been opened for that file have been closed. See Section 7.2.3.

The *path* is the filename of the file, including path information and optional device prefix. (See Section 7.2.5.)

Return Value

The function returns one of the following values:

0	if successful
-1	on failure

rename

Rename File

Syntax for C

#include {<stdio.h> | <file.h>}

int rename (const char *_old_name_ **, const char** *_new_name_ **);**

Syntax for C++

#include {<cstdio> | <file.h>}

int std::rename (const char *_old_name_ **, const char** *_new_name_ **);**

Description

The rename function changes the name of a file.

- The *old_name* is the current name of the file.
- The *new_name* is the new name for the file.

Note

The optional device specified in the new name must match the device of the old name. If they do not match, a file copy would be required to perform the rename, and rename is not capable of this action.

Return Value

The function returns one of the following values:

0	if successful
-1	on failure

Note

Although rename is a low-level function, it is defined by the C standard and can be used by portable applications.

Copyright © 2026 Texas Instruments Incorporated

7.2.3 Device-Driver Level I/O Functions

At the next level are the device-level drivers. They map directly to the low-level I/O functions. The default device driver is the HOST device driver, which uses the debugger to perform file operations. The HOST device driver is automatically used for the default C streams stdin, stdout, and stderr.

The HOST device driver shares a special protocol with the debugger running on a host system so that the host can perform the C I/O requested by the program. Instructions for C I/O operations that the program wants to perform are encoded in a special buffer named _CIOBUF_ in the .cio section. The debugger halts the program at a special breakpoint (C$$IO$$), reads and decodes the target memory, and performs the requested operation. The result is encoded into _CIOBUF_, the program is resumed, and the target decodes the result.

The HOST device is implemented with seven functions, HOSTopen, HOSTclose, HOSTread, HOSTwrite, HOSTlseek, HOSTunlink, and HOSTrename, which perform the encoding. Each function is called from the low-level I/O function with a similar name.

A device driver is composed of seven required functions. Not all function need to be meaningful for all devices, but all seven must be defined. Here we show the names of all seven functions as starting with DEV, but you may choose any name except for HOST.

Copyright © 2026 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. The file descriptor need not be unique across devices. The device file descriptor is used only by low-level functions when calling the device-driver-level functions. The low-level function `open` allocates its own unique file descriptor for the high-level functions to call the low-level functions. Code that uses only high-level I/O functions need not be aware of these file descriptors.

Copyright © 2026 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 *** *buf* **, unsigned** *count* **);**

Description

The read function reads *count* bytes from the input file associated with *dev_fd*.

- The *dev_fd* is the number assigned by open to an opened file.
- The *buf* is where the read characters are placed.
- The *count* is the number of characters to read from the file.

Return Value

This function must return -1 to indicate an error if for some reason no bytes could be read from the file. This could be because of an attempt to read from a O_WRONLY file, or for device-specific reasons.

If count is 0, no bytes are read and this function returns 0.

This function returns the number of bytes read, from 0 to count. 0 indicates that EOF was reached before any bytes were read. It is not an error to read less than count bytes; this is common if the are not enough bytes left in the file or the request was larger than an internal device buffer size.

Copyright © 2026 Texas Instruments Incorporated

DEV_write

Write Characters to a File

Syntax

int DEV_write (int *dev_fd* **, const char** **buf* **, unsigned***count* **);**

Description

This function writes *count* bytes to the output file.

- The *dev_fd* is the number assigned by open to an opened file.
- The *buffer* is where the write characters are placed.
- The *count* is the number of characters to write to the file.

Return Value

This function must return -1 to indicate an error if for some reason no bytes could be written to the file. This could be because of an attempt to read from a O_RDONLY file, or for device-specific reasons.

DEV_lseek

Set File Position Indicator

Syntax

off_t DEV_lseek (int *dev_fd* **, off_t** *offset* **, int** *origin* **);**

Description

This function sets the file's position indicator for this file descriptor as lseek.

If lseek is supported, it should not allow a seek to before the beginning of the file, but it should support seeking past the end of the file. Such seeks do not change the size of the file, but if it is followed by a write, the file size will increase.

Return Value

If successful, this function returns the new value of the file position indicator.

This function must return -1 to indicate an error if for some reason no bytes could be written to the file. For many devices, the lseek operation is nonsensical (e.g. a computer monitor).

DEV_unlink

Delete File

Syntax

int DEV_unlink (const char **path* **);**

Description

Remove the association of the pathname with the file. This means that the file may no longer be opened using this name, but the file may not actually be immediately removed.

Depending on the device, the file may be immediately removed, but for a device which allows open file descriptors to point to unlinked files, the file will not actually be deleted until the last file descriptor is closed. See Section 7.2.3.

Return Value

This function must return -1 to indicate an error if for some reason the file could not be unlinked (delayed removal does not count as a failure to unlink.)

If successful, this function returns 0.

Copyright © 2026 Texas Instruments Incorporated

DEV_rename

Rename File

Syntax

int DEV_rename (const char *_old_name_ **, const char** *_new_name_ **);**

Description

This function changes the name associated with the file.

- The *old_name* is the current name of the file.
- The *new_name* is the new name for the file.

Return Value

This function must return -1 to indicate an error if for some reason the file could not be renamed, such as the file does not exist, or the new name already exists.

Note

It is inadvisable to allow renaming a file so that it is on a different device. In general this would require a whole file copy, which may be more expensive than you expect.

If successful, this function returns 0.

Copyright © 2026 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.

Note

Use Unique Function Names

The function names open, read, write, close, lseek, rename, and unlink are used by the low-level routines. Use other names for the device-level functions that you write.

Use the low-level function add_device() to add your device to the device_table. The device table is a statically defined array that supports *n* devices, where *n* is defined by the macro _NDEVICE found in stdio.h/cstdio.

The first entry in the device table is predefined to be the host device on which the debugger is running. The low-level routine add_device() finds the first empty position in the device table and initializes the device fields with the passed-in arguments. For a complete description, see the add_device function.

Example 7-1. Mapping Default Streams to Device

```
#include <stdio.h>
#include <file.h>
#include "mydevice.h"
void main()
{
    add_device("mydevice", _MSA,
               MYDEVICE_open, MYDEVICE_close,
               MYDEVICE_read, MYDEVICE_write,
               MYDEVICE_lseek, MYDEVICE_unlink, MYDEVICE_rename);
    /*--------------------------------------------------------------------*/
    /* Re-open stderr as a MYDEVICE file                                  */
    /*--------------------------------------------------------------------*/
    if (!freopen("mydevice:stderrfile", "w", stderr))
    {
        puts("Failed to freopen stderr");
        exit(EXIT_FAILURE);
    }
    /*--------------------------------------------------------------------*/
    /* stderr should not be fully buffered; we want errors to be seen as  */
    /* soon as possible.  Normally stderr is line-buffered, but this example */
    /* does not buffer stderr at all. This means that there will be one call */
    /* to write() for each character in the message.                      */
    /*--------------------------------------------------------------------*/
    if (setvbuf(stderr, NULL, _IONBF, 0))
    {
        puts("Failed to setvbuf stderr");
        exit(EXIT_FAILURE);
    }
    /*--------------------------------------------------------------------*/
    /* Try it out!                                                        */
    /*--------------------------------------------------------------------*/
    printf("This goes to stdout\n");
    fprintf(stderr, "This goes to stderr\n"); }
```


Copyright © 2026 Texas Instruments Incorporated

7.2.5 The device Prefix

A file can be opened to a user-defined device driver by using a device prefix in the pathname. The device prefix is the device name used in the call to add_device followed by a colon. For example:

```
FILE *fptr = fopen("mydevice:file1", "r");
int fd = open("mydevice:file2, O_RDONLY, 0);
```

If no device prefix is used, the HOST device will be used to open the file.

add_device

Add Device to Device Table

Syntax for C

#include <file.h>

int add_device(char * *name*,
unsigned *flags* **,**
int (* *dopen*)**(const char** **path*, **unsigned** *flags*, **int llv_fd),**
int (* *dclose* **)(int dev_fd),**
int (* *dread*)**(int***dev_fd*, **char** **buf*, **unsigned** *count*),
int (* *dwrite* **)(int** *dev_fd*, **const char** **buf*, **unsigned** *count*),
off_t (* *dlseek* **)(int dev_fd, off_t** *ioffset*, **int** *origin*),
int (* *dunlink* **)(const char** **path*),
int (* *drename* **)(const char** **old_name*, **const char** **new_name*));

Defined in

lowlev.c (in the lib/src subdirectory of the compiler installation)

Description

The add_device function adds a device record to the device table allowing that device to be used for I/O from C. The first entry in the device table is predefined to be the HOST device on which the debugger is running. The function add_device() finds the first empty position in the device table and initializes the fields of the structure that represent a device.

To open a stream on a newly added device use fopen( ) with a string of the format *devicename* **:***filename* as the first argument.

- The *name* is a character string denoting the device name. The name is limited to 8 characters.
- The *flags* are device characteristics. The flags are as follows:

  _SSA Denotes that the device supports only one open stream at a time

  _MSA Denotes that the device supports multiple open streams

  More flags can be added by defining them in file.h.
- The *dopen*, *dclose*, *dread*, *dwrite*, *dlseek*, *dunlink*, and *drename* specifiers are function pointers to the functions in the device driver that are called by the low-level functions to perform I/O on the specified device. You must declare these functions with the interface specified in Section 7.2.2. The device driver for the HOST that the C7000 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 © 2026 Texas Instruments Incorporated

add_device (continued)

Add Device to Device Table

Example

Example 7-2 does the following:

- Adds the device *mydevice* to the device table
- Opens a file named *test* on that device and associates it with the FILE pointer *fid*
- Writes the string *Hello, world* into the file
- Closes the file

Example 7-2 illustrates adding and using a device for C I/O:

Example 7-2. Program for C I/O Device

```
#include <file.h>
#include <stdio.h>
/*****************************************************************************/
/* Declarations of the user-defined device drivers                           */
/*****************************************************************************/
extern int   MYDEVICE_open(const char *path, unsigned flags, int fno);
extern int   MYDEVICE_close(int fno);
extern int   MYDEVICE_read(int fno, char *buffer, unsigned count);
extern int   MYDEVICE_write(int fno, const char *buffer, unsigned count);
extern off_t MYDEVICE_lseek(int fno, off_t offset, int origin);
extern int   MYDEVICE_unlink(const char *path);
extern int   MYDEVICE_rename(const char *old_name, char *new_name);
main()
{
   FILE *fid;
   add_device("mydevice", _MSA, MYDEVICE_open, MYDEVICE_close, MYDEVICE_read,
              MYDEVICE_write, MYDEVICE_lseek, MYDEVICE_unlink, MYDEVICE_rename);
   fid = fopen("mydevice:test","w");
   fprintf(fid,"Hello, world\n");

   fclose(fid);
}
```


Copyright © 2026 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, systems that integrate an RTOS typically have multiple threads that 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 ; 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, 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 RTOS kernels are responsible for creating and installing the primitives, so you do not need to take any action. However, this mechanism can be used in multi-threaded applications that do not use such locking mechanisms.

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 © 2026 Texas Instruments Incorporated

7.4 Library-Build Process

When using the C/C++ compiler, you can compile your code under a number of different configurations and options that are not necessarily compatible with one another. Because it would be infeasible to include all possible run-time-support library variants, compiler releases pre-build only a small number of very commonly-used libraries such as rts7100_le.lib.

To provide maximum flexibility, the run-time-support source code is provided as part of each compiler release. You can build the missing libraries as desired. The linker can also automatically build missing libraries. This is accomplished with a new library build process, the core of which is the executable mklib, which is available beginning with CCS 5.1.

7.4.1 Required Non-Texas Instruments Software

To use the self-contained run-time-support build process to rebuild a library with custom options, the following are required:

- sh (Bourne shell)
- gmake (GNU make 3.81 or later)

  More information is available from GNU at http://www.gnu.org/software/make. GNU make (gmake) is also available in earlier versions of Code Composer Studio. GNU make is also included in some UNIX support packages for Windows, such as the MKS Toolkit, Cygwin, and Interix. The GNU make used on Windows platforms should explicitly report "This program built for Windows32" when the following is executed from the Command Prompt window:

```
gmake -h
```

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 C7X_C_DIR environment variable. Typically, one of the pathnames in C7X_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 C7X_C_DIR to find a library that is the best match for the build attributes of the application. The build attributes are set indirectly according to the command-line options used to build the application. Build attributes include things like CPU revision. If the library name is explicitly specified (e.g. --library=rts7100_le.lib), run-time support looks for that library exactly. If the library name is not specified, the linker uses the index library libc.a to pick an appropriate library. If the library is specified by path (e.g. --library=/foo/rts7100_le.lib), it is assumed the library already exists and it will not be built automatically.

The index library describes a set of libraries with different build attributes. The linker will compare the build attributes for each potential library with the build attributes of the application and will pick the best fit. For details on the index library, see Chapter 10.

Copyright © 2026 Texas Instruments Incorporated

Now that the linker has decided which library to use, it checks whether the run-time-support library is present in C7X_C_DIR. The library must be in exactly the same directory as the index library libc.a. If the library is not present, the linker invokes mklib to build it. This happens when the library is missing, regardless of whether the user specified the name of the library directly or allowed the linker to pick the best library from the index library.

The mklib program builds the requested library and places it in 'lib' directory part of C7X_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=rts7100_le.lib
```

7.4.2.2.2 Shared or Read-Only Library Directory

If the compiler tools are to be installed in shared or read-only directory, mklib cannot build the standard libraries at link time; the libraries must be built before the library directory is made shared or read-only.

At installation time, the installing user must build all of the libraries which will be used by any user. To build all possible libraries, change the working directory to be the compiler RTS library directory 'lib' and invoke the mklib executable there:

```
mklib --all
```

Some targets have many libraries, so this step can take a long time. To build a subset of the libraries, invoke mklib individually for each desired library.

7.4.2.2.3 Building Libraries With Custom Options

You can build a library with any extra custom options desired. This is useful for building a version of the library with silicon exception workarounds enabled. The generated library is not a standard library, and must not be

Copyright © 2026 Texas Instruments Incorporated

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 rts7100_le.lib, change the working directory to the 'lib' directory and run the command:

```
mklib --pattern=rts7100_le.lib --name=rts7100_le_debug.lib --install_to=$Project/Debug
  --extra_options=<options_list>
```

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 (in the lib/src subdirectory of the compiler installation). REQUIRED.
--pattern=*filename*	Pattern for building a library. If neither --extra_options nor --options are specified, the library will be the standard library with the standard options for that library. If either --extra_options or --options are specified, the library is a custom library with custom options. REQUIRED unless --all is used.
--all	Build all standard libraries at once.
--install_to=*directory*	The directory into which to write the library. For a standard library, this defaults to the same directory as the index library (libc.a). For a custom library, this option is REQUIRED.
--compiler_bin_dir= *directory*	The directory where the compiler executables are. When invoking mklib directly, the executables should be in the path, but if they are not, this option must be used to tell mklib where they are. This option is primarily for use when mklib is invoked by the linker.
--name=*filename*	File name for the library with no directory part. Only useful for custom libraries.
--options='*str***'**	Options to use when building the library. The default options (see below) are *replaced* by this string. If this option is used, the library will be a custom library.
--extra_options='*str***'**	Options to use when building the library. The default options (see below) are also used. If this option is used, the library will be a custom library.
--list_libraries	List the libraries this script is capable of building and exit. ordinary system-specific directory.
--log=*filename*	Save the build log as *filename*.
--tmpdir=*directory*	Use *directory* for scratch space instead of the ordinary system-specific directory.
--gmake=*filename*	Gmake-compatible program to invoke instead of "gmake"
--parallel=*N*	Compile *N* files at once ("gmake -j N").
--query=*filename*	Does this script know how to build FILENAME?
--help or **--h**	Display this help.
--quiet or **--q**	Operate silently.
--verbose or **--v**	Extra information to debug this executable.

Examples:

To build all standard libraries and place them in the compiler's library directory:

```
mklib --all --index=$C_DIR/lib
```

To build one standard library and place it in the compiler's library directory:

```
mklib --pattern=rts7100_le.lib --index=$C_DIR/lib
```

To build a custom library that is just like rts7100_le.lib, but has symbolic debugging support enabled:

```
mklib --pattern=rts7100_le.lib --extra_options="-g" --index=$C_DIR/lib --install_to=$Project/Debug
--name=rts7100_le_debug.lib
```


Copyright © 2026 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 that knows how to use the files in the lib/src subdirectory of the compiler installation and invoke gmake with the appropriate options to build each library. If necessary, mklib can be bypassed and the Makefile used directly, but this mode of operation is not supported by TI, and you are responsible for any changes to the Makefile. The format of the Makefile and the interface between mklib and the Makefile is subject to change without notice. The mklib program is the forward-compatible path.

7.4.3.2 Libraries From Other Vendors

Any vendor who wishes to distribute a library that can be rebuilt automatically by the linker must provide:

- An index library (like 'libc.a', but with a different name)
- A copy of mklib specific to that library
- A copy of the library source code (in whatever format is convenient)

These things must be placed together in one directory that is part of the linker's library search path (specified either in C7X_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 © 2026 Texas Instruments Incorporated

Chapter 8

Introduction to Object Modules

The linker creates executable object files from object modules. These executable object files can be executed by a C7000 CPU.

Object modules make modular programming easier because they encourage you to think in terms of *blocks* of code and data. These blocks are known as sections. The linker provides directives that allow you to create and manipulate sections.

This chapter focuses on the concept and use of sections.

8.1 Object File Format Specifications.....201
8.2 Executable Object Files.....201
8.3 Introduction to Sections.....201
8.4 How the Linker Handles Sections.....202
8.5 Symbols.....203
8.6 Loading a Program.....204

Copyright © 2026 Texas Instruments Incorporated

8.1 Object File Format Specifications

The object files created by the Code Generation Tools conform to the ELF (Executable and Linking Format) binary format, which is used by the Embedded Application Binary Interface (EABI).

The C7000 has a 48-bit address space for both code and data. Accordingly, object files use the ELF64 format.

8.2 Executable Object Files

The linker can be used to produce static executable object modules. An executable object module has the same format as object files that are used as linker input. The sections in an executable object module, however, have been combined and placed in target memory, and the relocations are all resolved.

To run a program, the data in the executable object module must be transferred, or loaded, into target system memory. See Chapter 9 for details about loading and running programs.

8.3 Introduction to Sections

The smallest unit of an object file is a *section*. A section is a block of code or data that occupies contiguous space in the memory map. Each section of an object file is separate and distinct.

ELF format executable object files contain *segments*. An ELF segment is a meta-section. It represents a contiguous region of target memory. It is a collection of *sections* that have the same property, such as writeable or readable. An ELF loader needs the segment information, but does not need the section information. The ELF standard allows the linker to omit ELF section information entirely from the executable object file.

Object files usually contain three default sections:

.text section	Contains executable code [1]
.data section	Usually contains initialized data
.bss	Usually reserves space for uninitialized variables

The linker allows you to create, name, and link other kinds of sections. The .text, .data, and .bss sections are archetypes for how sections are handled.

There are two basic types of sections:

Initialized sections	Contain data or code. The .text and .data sections are initialized.
Uninitialized sections	Reserve space in the memory map for uninitialized data. The .bss section is uninitialized.

One of the linker's functions is to relocate sections into the target system's memory map; this function is called *placement*. Because most systems contain several types of memory, using sections can help you use target memory more efficiently. All sections are independently relocatable; you can place any section into any allocated block of target memory. For example, you can define a section that contains an initialization routine and then allocate the routine in a portion of the memory map that contains ROM. For information on section placement, see Section 11.3.5.

[1] Some targets allow content other than text, such as constants, in .text sections.

Copyright © 2026 Texas Instruments Incorporated

Figure 8-1 shows the relationship between sections in an object file and a hypothetical target memory. ROM may be EEPROM, FLASH or some other type of physical memory in an actual system.

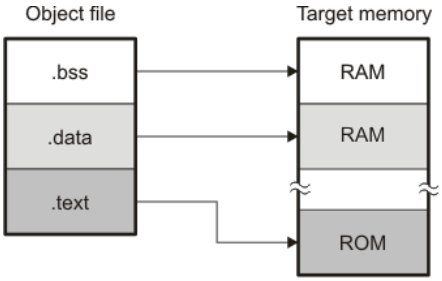


Figure 8-1. Partitioning Memory Into Logical Blocks

8.3.1 Special Section Names

You can use the .sect and .usect directives to create any section name you like, but certain sections are treated in a special manner by the linker and the compiler's run-time support library. If you create a section with the same name as a special section, you should take care to follow the rules for that special section.

A few common special sections are:

- .text -- Used for program code.
- .data -- Used for initialized non-const objects (global variables).
- .bss -- Used for uninitialized objects (global variables).
- .const -- Used for initialized const objects (string constants, variables declared const).
- .cinit -- Used to initialize C global variables at startup.
- .stack -- Used for the function call stack.
- .sysmem - Used for the dynamic memory allocation pool.

For more information on sections, see Section 11.3.5.

8.4 How the Linker Handles Sections

The linker has two main functions related to sections. First, the linker uses sections in object files as building blocks; it combines input sections to create output sections in an executable output module. Second, the linker chooses memory addresses for output sections; this is called *placement*. Two linker directives support these functions:

- The *MEMORY* directive allows you to define the memory map of a target system. You can name portions of memory and specify their starting addresses and their lengths.
- The *SECTIONS* directive tells the linker how to combine input sections into output sections and where to place these output sections in memory.

Subsections let you manipulate the placement of sections with greater precision. You can specify the location of each subsection with the linker's SECTIONS directive. If you do not specify a subsection, the subsection is combined with the other sections with the same base section name. See Section 12.5.5.1.

It is not always necessary to use linker directives. If you do not use them, the linker uses the target processor's default placement algorithm described in Section 12.7. When you *do* use linker directives, you must specify them in a linker command file.

Refer to the following sections for more information about linker command files and linker directives:

- Section 12.5, *Linker Command Files*
- Section 12.5.4, *The MEMORY Directive*
- Section 12.5.5, *The SECTIONS Directive*
- Section 12.7, *Default Placement Algorithm*

Copyright © 2026 Texas Instruments Incorporated

8.4.1 Combining Input Sections

Figure 8-2 provides a simplified example of the process of linking two files together.

Note that this is a simplified example, so it does not show all the sections that will be created or the actual sequence of the sections. See Section 12.7 for the actual default memory placement map for C7000.

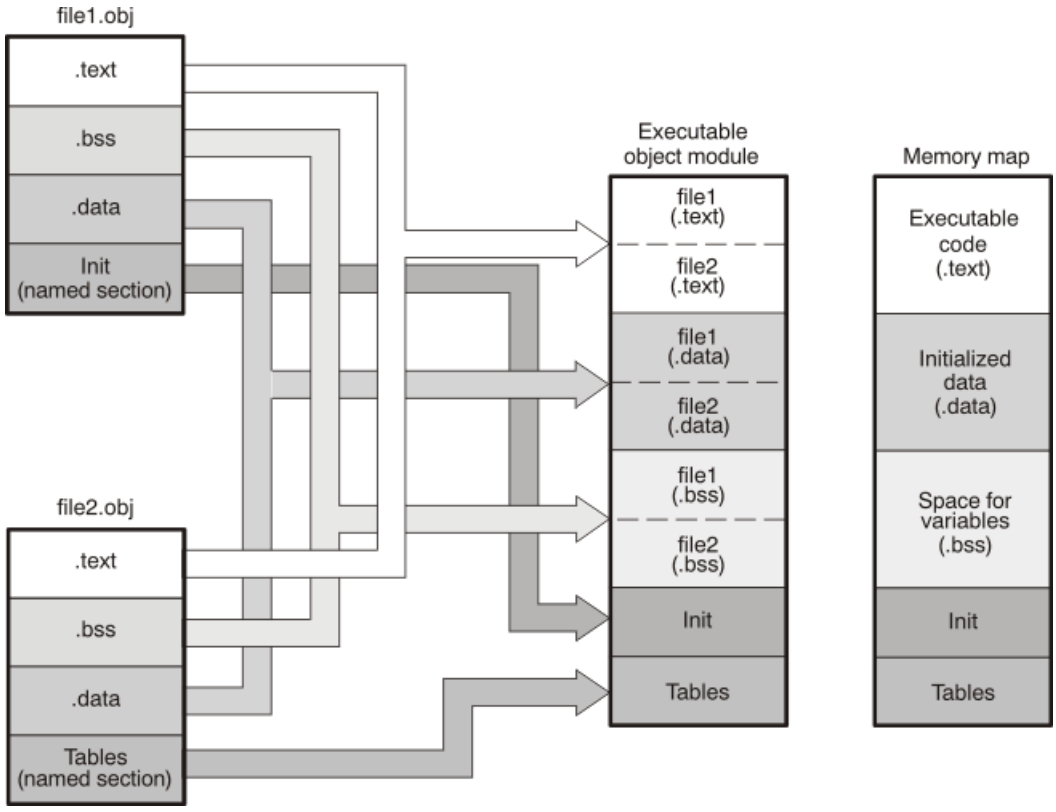


Figure 8-2. Combining Input Sections to Form an Executable Object Module

In Figure 8-2, file1.obj and file2.obj have been assembled to be used as linker input. Each contains the .text, .data, and default sections; in addition, each contains a user-named section. The executable object module shows the combined sections. The linker combines the .text section from file1.obj and the .text section from file2.obj to form one .text section, then combines the two .data sections and the two sections, and finally places the user-named sections at the end. The memory map shows the combined sections to be placed into memory.

8.4.2 Placing Sections

Figure 8-2 illustrates the linker's default method for combining sections. Sometimes you may not want to use the default setup. For example, you may not want all of the .text sections to be combined into a single .text section. Or you may want a user-named section placed where the .data section would normally be allocated. Most memory maps contain various types of memory (RAM, ROM, EEPROM, FLASH, etc.) in varying amounts; you may want to place a section in a specific type of memory.

For further explanation of section placement within the memory map, see the discussions in Section 12.5.4 and Section 12.5.5. See Section 12.7 for the actual default memory allocation map for C7000.

8.5 Symbols

An object file contains a symbol table that stores information about *symbols* in the object file. The linker uses this table when it performs relocation.

Copyright © 2026 Texas Instruments Incorporated

An object file symbol is a named 48-bit integer value, usually representing an address. A symbol can represent things like the start address of a function, variable, section, or an absolute integer (such as the size of the stack).

Absolute symbols are symbols with a numeric value. They may be constants. To the linker, such symbols are unsigned, but the integer may be treated as signed or unsigned depending on how it is used. The range of legal values for an absolute integer is 0 to 2^48-1 for unsigned treatment and -2^47 to 2^47-1 for signed treatment.

8.5.1 Local Symbols

Local symbols are visible within a single object file. Each object file may have its own local definition for a particular symbol. References to local symbols in an object file are entirely unrelated to local symbols of the same name in another object file. By default, a symbol is local.

8.5.2 Weak Symbols

Weak symbols are symbols that may or may not be defined.

The linker processes symbols that are defined with a "weak" binding differently from symbols that are defined with global binding. Instead of including a weak symbol in the object file's symbol table (as it would for a global symbol), the linker only includes a weak symbol in the output of a "final" link if the symbol is required to resolve an otherwise unresolved reference.

This allows the linker to minimize the number of symbols it includes in the output file's symbol table by omitting those that are not needed to resolve references. Reducing the size of the output file's symbol table reduces the time required to link, especially if there are a large number of pre-loaded symbols to link against.

- **Using the Linker Command File:** To define a weak symbol in a linker command file, use the "weak" operator in an assignment expression to designate that the symbol as eligible for removal from the output file's symbol table if it is not referenced. In a linker command file, an assignment expression outside a MEMORY or SECTIONS directive can be used to define a weak linker-defined symbol. For example, you can define "ext_addr_sym" as follows:

  ```
  weak(ext_addr_sym) = 0x12345678;
  ```

  If the linker command file performs the final link, then "ext_addr_sym" is presented to the linker as a weak symbol; it will not be included in the resulting output file if the symbol is not referenced. See Section 12.6.6.
- **Using C/C++ code:** See information about the WEAK pragma and weak GCC-style variable attribute in Section 5.8.35.

If there are multiple definitions of the same symbol, the linker uses certain rules to determine which definition takes precedence. Some definitions may have weak binding and others may have strong binding. "Strong" in this context means that the symbol has *not* been given a weak binding in an assignment statement in a linker command file. The linker uses the following guidelines to determine which definition is used when resolving references to a symbol:

- A strongly bound symbol always takes precedence over a weakly bound symbol.
- If two symbols are both strongly bound or both weakly bound, a symbol defined in a linker command file takes precedence over a symbol defined in an input object file.
- If two symbols are both strongly bound and both are defined in an input object file, the linker provides a symbol redefinition error and halts the link process.

8.6 Loading a Program

The linker creates an executable object file which can be loaded in several ways, depending on your execution environment. These methods include using Code Composer Studio or a hex conversion utility. For details, see Section 9.1.

Copyright © 2026 Texas Instruments Incorporated

Chapter 9

Program Loading and Running

Even after a program is written, compiled, and linked into an executable object file, there are still many tasks that need to be performed before the program does its job. The program must be loaded onto the target, memory and registers must be initialized, and the program must be set to running.

Some of these tasks need to be built into the program itself. Many of the necessary tasks are handled for you by the compiler and linker, but if you need more control over these tasks, it helps to understand how the pieces are expected to fit together.

This chapter will introduce you to the concepts involved in program loading, initialization, and startup.

Refer to your device documentation for various device-specific aspects of bootstrapping.

9.1 Loading 206
9.2 Entry Point 206
9.3 Run-Time Initialization 207
9.4 Arguments to main 210
9.5 Run-Time Relocation 210
9.6 Additional Information 210

Copyright © 2026 Texas Instruments Incorporated

9.1 Loading

A program needs to be placed into the target device's memory before it may be executed. *Loading* is the process of preparing a program for execution by initializing device memory with the program's code and data. A *loader* might be another program on the device, an external agent (for example, a debugger), or the device might initialize itself after power-on, which is known as *bootstrap loading*, or *bootloading*.

The loader is responsible for constructing the *load image* in memory before the program starts. The load image is the program's code and data in memory before execution. What exactly constitutes loading depends on the environment, such as whether an operating system is present. This section describes several loading schemes for bare-metal devices. This section is not exhaustive.

A program may be loaded in the following ways:

- **A debugger running on a connected host workstation.** In a typical embedded development setup, the device is subordinate to a host running a debugger such as Code Composer Studio (CCS). The device is connected with a communication channel such as a JTAG interface. CCS reads the program and writes the load image directly to target memory through the communications interface.
- **Loader running on another CPU.** Using an ELF loader, another CPU on the device (typically an ARM core) loads the executable into memory.
- **Another program running on the device.** The running program can create the load image and transfer control to the loaded program. If an operating system is present, it may have the ability to load and run programs.

9.2 Entry Point

The entry point is the address at which the execution of the program begins. This is the address of the startup routine. The startup routine is responsible for initializing and calling the rest of the program. For a C/C++ program, the startup routine is usually named _c_int00 (see Section 9.3.1). After the program is loaded, the value of the entry point is placed in the PC register and the CPU is allowed to run.

The object file has an entry point field. For a C/C++ program, the linker will fill in _c_int00 by default. You can select a custom entry point; see Section 12.4.11. The device itself cannot read the entry point field from the object file, so it has to be encoded in the program somewhere.

- If you are using a bootloader, the boot table includes an entry point field. When it finishes running, the bootloader branches to the entry point.
- If you are using a hosted debugger, such as CCS, the debugger may explicitly set the program counter (PC) to the value of the entry point.

Copyright © 2026 Texas Instruments Incorporated

9.3 Run-Time Initialization

After the load image is in place, the program can run. The subsections that follow describe bootstrap initialization of a C/C++ program.

9.3.1 The _c_int00 Function

The function _c_int00 is the *startup routine* (also called the *boot routine*) for C/C++ programs. It performs all the steps necessary for a C/C++ program to initialize itself.

The name _c_int00 means that it is the interrupt handler for interrupt number 0, RESET, and that it sets up the C environment. Its name need not be exactly _c_int00, but the linker sets _c_int00 as the entry point for C programs by default. The compiler's run-time-support library provides a default implementation of _c_int00.

The startup routine is responsible for performing the following actions:

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

9.3.2 RAM Model vs. ROM Model

Choose a startup model based on the needs of your application. The ROM model performs more work during the boot routine. The RAM model performs more work while loading the application.

If your application is likely to need frequent RESETs or is a standalone application, the ROM model may be a better choice, because the boot routine will have all the data it needs to initialize RAM variables. However, for a system with an operating system, it may be better to use the RAM model.

C boot routine copies data from the .cinit section to the run-time location of the variables to be initialized.

9.3.2.1 Autoinitializing Variables at Run Time (--rom_model)

Autoinitializing variables at run time is the default method of autoinitialization. To use this method, invoke the linker with the --rom_model option.

The ROM model allows initialization data to be stored in slow non-volatile memory and copied to fast memory each time the program is reset. Use this method if your application runs from code burned into slow memory or needs to survive a reset.

For the ROM model , the .cinit section is loaded into memory along with all the other initialized sections. The linker defines a special symbol called __TI_CINIT_Base that points to the beginning of the initialization tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by .cinit) into the run-time location of the variables.

Copyright © 2026 Texas Instruments Incorporated

Figure 9-1 illustrates autoinitialization at run time using the ROM model.

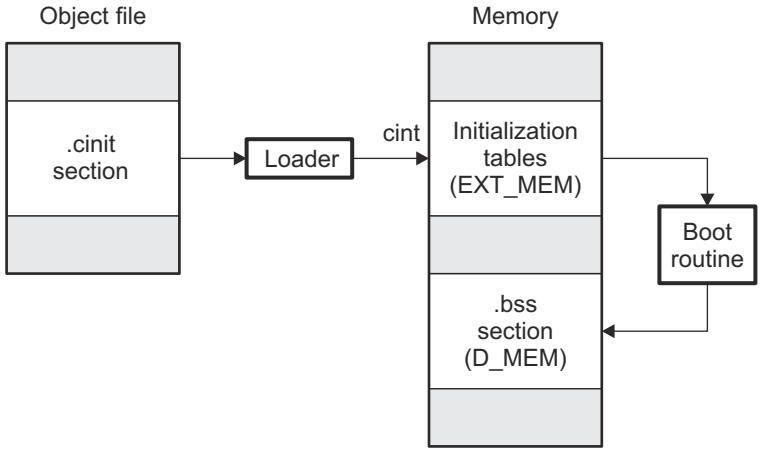


Figure 9-1. Autoinitialization at Run Time

9.3.2.2 Initializing Variables at Load Time (--ram_model)

The RAM model Initializes variables at load time. To use this method, invoke the linker with the --ram_model option.

This model may reduce boot time and save memory used by the initialization tables.

When you use the --ram_model linker 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 sets __TI_CINIT_Base equal to __TI_CINIT_Limit to indicate there are no .cinit records.

The loader copies values directly from the .data section to memory.

Figure 9-2 illustrates the initialization of variables at load time.

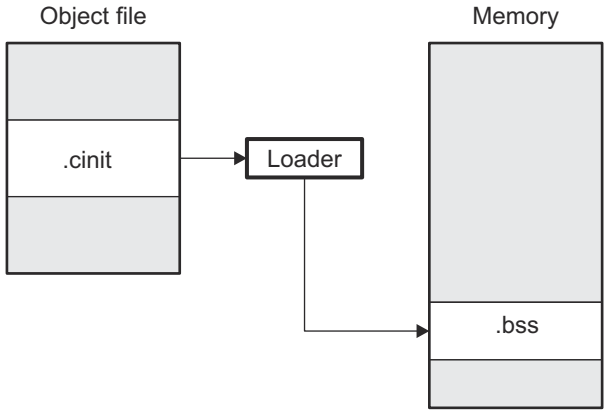


Figure 9-2. Initialization at Load Time

Copyright © 2026 Texas Instruments Incorporated

9.3.2.3 The --rom_model and --ram_model Linker Options

The following list outlines what happens when you invoke the linker with the --ram_model or --rom_model option.

- The symbol _c_int00 is defined as the program entry point. The _c_int00 symbol is the start of the C boot routine in boot.c.obj. Referencing _c_int00 ensures that boot.c.obj is automatically linked in from the appropriate run-time-support library.
- When you use the ROM model to autoinitialize at run time (--rom_model option):
  - The linker defines a special symbol called __TI_CINIT_Base that points to the beginning of the initialization tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by .cinit) into the run-time location of the variables.
- When you use the RAM model to initialize at load time (--ram_model option):
  - The linker sets __TI_CINIT_Base equal to __TI_CINIT_Limit to indicate there are no .cinit records.

9.3.3 About Linker-Generated Copy Tables

The RTS function copy_in can be used at run-time to move code and data around, usually from its load address to its run address. This function reads size and location information from copy tables. The linker automatically generates several kinds of copy tables. Refer to Section 12.8.

You can create and control code overlays with copy tables. See Section 12.8.4 for details and examples.

Copy tables can be used by the linker to implement run-time relocations as described in Section 9.5, however copy tables require a specific table format.

9.3.3.1 BINIT

The BINIT (boot-time initialization) copy table is special in that the target will automatically perform the copying at auto-initialization time. Refer to Section 12.8.4.2 for more about the BINIT copy table name. The BINIT copy table is copied before .cinit processing.

9.3.3.2 CINIT

EABI .cinit tables are special kinds of copy tables. Refer to Section 9.3.2.1 for more about using the .cinit section with the ROM model and Section 9.3.2.2 for more using it with the RAM model.

Copyright © 2026 Texas Instruments Incorporated

9.4 Arguments to main

Some programs expect arguments to main (argc, argv) to be valid. Normally this is not possible for an embedded program, but the TI runtime does provide a way to do it. The user must allocate an .args section of an appropriate size using the --args linker option. It is the responsibility of the loader to populate the .args section. It is not specified how the loader determines which arguments to pass to the target. The format of the arguments is the same as an array of pointers to char on the target.

See Section 12.4.4 for information about allocating memory for argument passing.

9.5 Run-Time Relocation

At times you may want to load code into one area of memory and move it to another area before running it. For example, you may have performance-critical code in an external-memory-based system. The code must be loaded into external memory, but it would run faster in internal memory. Because internal memory is limited, you might swap in different speed-critical functions at different times.

The linker provides a way to handle this. Using the SECTIONS directive, you can optionally direct the linker to allocate a section twice: first to set its load address and again to set its run address. Use the *load* keyword for the load address and the *run* keyword for the run address. If a section is assigned two addresses at link time, all labels defined in the section are relocated to refer to the run-time address so that references to the section (such as branches) are correct when the code runs.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and loads and runs at the same address. If you provide both allocations, the section is actually allocated as if it were two separate sections. The two sections are the same size if the load section is not compressed.

Uninitialized sections (such as) are not loaded, so the only significant address is the run address. The linker allocates uninitialized sections only once; if you specify both run and load addresses, the linker warns you and ignores the load address.

For a complete description of run-time relocation, see Section 12.5.6.

9.6 Additional Information

See the following sections and documents for additional information:

- Section 6.7, "System Initialization"
- Section 11.3.2, "Run-Time Initialization"
- Section 12.4.4, "Allocate Memory for Use by the Loader to Pass Arguments (--arg_size Option)"
- Section 12.4.11, "Define an Entry Point (--entry_point Option)"
- Section 12.5.6.1 ,"Specifying Load and Run Addresses"
- Section 12.8, "Linker-Generated Copy Tables"
- Section 12.10.1, "Run-Time Initialization"

Copyright © 2026 Texas Instruments Incorporated

Chapter 10

Archiver Description

The C7000 archiver lets you combine several individual files into a single archive file. You can use the archiver to collect a group of object files into an object library. The linker includes in the library the members that resolve external references during the link. The archiver allows you to modify a library by deleting, replacing, extracting, or adding members.

10.1 Archiver Overview 212
10.2 The Archiver's Role in the Software Development Flow 212
10.3 Invoking the Archiver 213
10.4 Archiver Examples 214
10.5 Library Information Archiver Description 215

Copyright © 2026 Texas Instruments Incorporated

10.1 Archiver Overview

You can build libraries from any type of files. The linker accepts archive libraries as input. It can use libraries that contain individual object files.

One of the most useful applications of the archiver is building libraries of object modules. For example, you can write several arithmetic routines, assemble them, and use the archiver to collect the object files into a single, logical group. You can then specify the object library as linker input. The linker searches the library and includes members that resolve external references.

10.2 The Archiver's Role in the Software Development Flow

Figure 10-1 shows the archiver's role in the software development process. The shaded portion highlights the most common archiver development path.

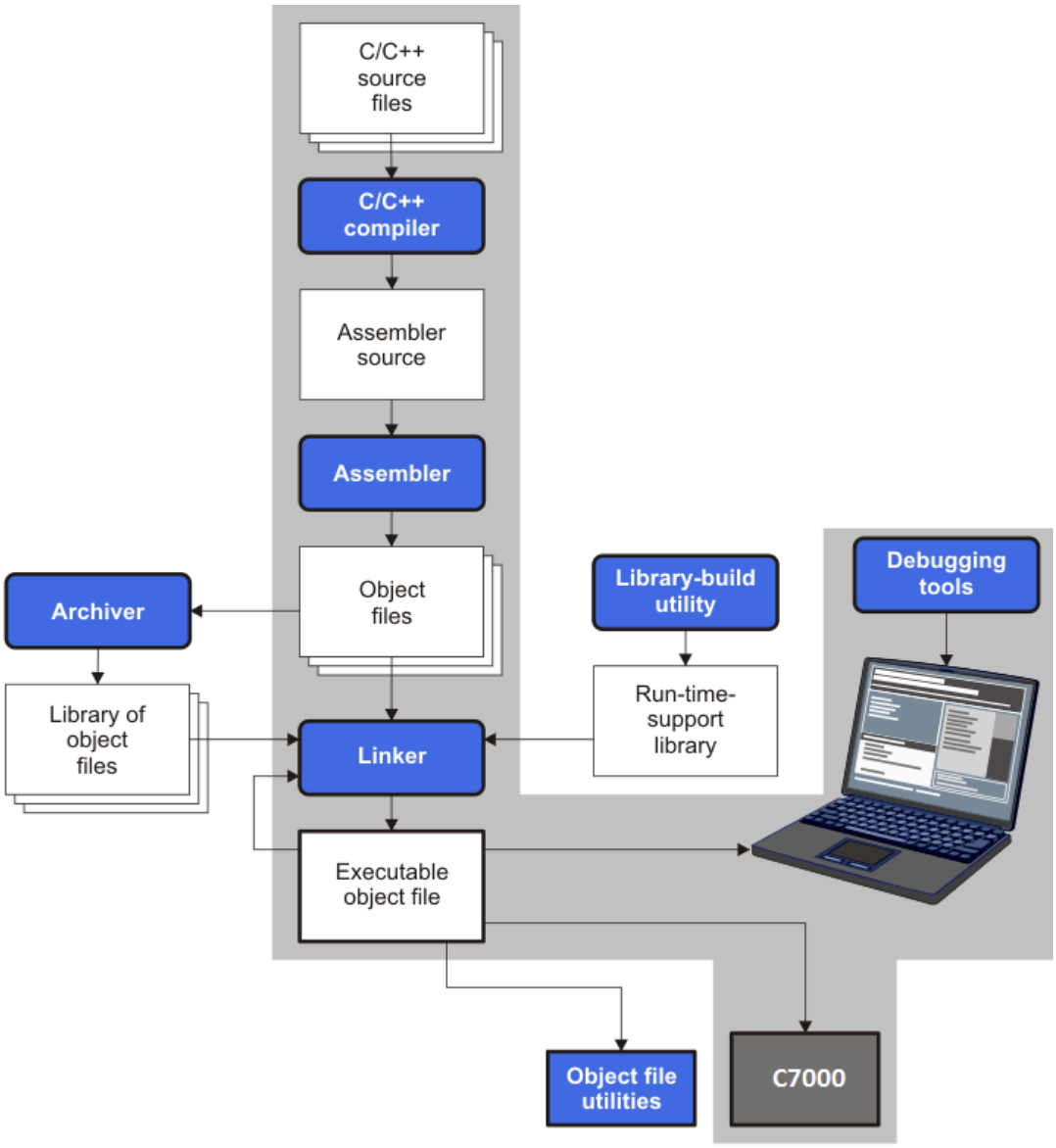


Figure 10-1. The Archiver in the C7000 Software Development Flow

Copyright © 2026 Texas Instruments Incorporated

10.3 Invoking the Archiver

To invoke the archiver, enter:

ar7x[-]*command* [*options*] *libname* [*filename*$_1$ ... *filename*$_n$]

ar7x is the command that invokes the archiver.

[-]*command* tells the archiver how to manipulate the existing library members and any specified. A command can be preceded by an optional hyphen. You must use one of the following commands when you invoke the archiver, but you can use only one command per invocation. The archiver commands are as follows:

- **@** uses the contents of the specified file instead of command line entries. You can use this command to avoid limitations on command line length imposed by the host operating system. Use a ; at the beginning of a line in the command file to include comments. (See Archiver Command File for an example using an archiver command file.)
- **a** adds the specified files to the library. This command does not replace an existing member that has the same name as an added file; it simply *appends* new members to the end of the archive.
- **d** deletes the specified members from the library.
- **r** replaces the specified members in the library. If you do not specify filenames, the archiver replaces the library members with files of the same name in the current directory. If the specified file is not found in the library, the archiver adds it instead of replacing it.
- **t** prints a table of contents of the library. If you specify filenames, only those files are listed. If you do not specify any filenames, the archiver lists all the members in the specified library.
- **x** extracts the specified files. If you do not specify member names, the archiver extracts all library members. When the archiver extracts a member, it simply copies the member into the current directory; it *does not* remove it from the library.

options In addition to one of the *commands,* you can specify options. To use options, combine them with a command; for example, to use the a command and the s option, enter -as or as. The hyphen is optional for archiver options only. These are the archiver options:

- **-h** provide command-line help
- **-q** (quiet) suppresses the banner and status messages.
- **-s** prints a list of the global symbols that are defined in the library. (This option is valid only with the a, r, and d commands.)
- **-u** replaces library members only if the replacement has a more recent modification date. You must use the r command with the -u option to specify which members to replace.
- **-v** (verbose) provides a file-by-file description of the creation of a new library from an old library and its members.

libname names the archive library to be built or modified. If you do not specify an extension for *libname,* the archiver uses the default extension *.lib*.

filenames names individual files to be manipulated. These files can be existing library members or new files to be added to the library. When you enter a filename, you must enter a complete filename including extension, if applicable.

Note

Naming Library Members: It is possible (but not desirable) for a library to contain several members with the same name. If you attempt to delete, replace, or extract a member whose name is the same as another library member, the archiver deletes, replaces, or extracts the first library member with that name.

Copyright © 2026 Texas Instruments Incorporated

10.4 Archiver Examples

The following are examples of typical archiver operations:

- If you want to create a library called function.lib that contains the files sine.obj, cos.obj, and flt.obj, enter:

```
ar7x -a function sine.obj cos.obj flt.obj
```

  The archiver responds as follows:

```
==> new archive 'function.lib'
==> building new archive 'function.lib'
```

- You can print a table of contents of function.lib with the -t command, enter:

```
ar7x -t function
```

  The archiver responds as follows:

```
    SIZE   DATE                        FILE NAME
--------   ------------------------    -----------------
    4260   Thu Mar 28 15:38:18 2019    sine.obj
    4260   Thu Mar 28 15:38:18 2019    cos.obj
    4260   Thu Mar 28 15:38:18 2019    flt.obj
```

- If you want to add new members to the library, enter:

```
ar7x -as function atan.obj
```

  The archiver responds as follows:

```
    ==> symbol defined: '_sin'
    ==> symbol defined: '_cos'
    ==> symbol defined: '_tan'
    ==> symbol defined: '_atan
    ==> building archive 'function.lib'
```

  Because this example does not specify an extension for the libname, the archiver adds the files to the library called function.lib. If function.lib does not exist, the archiver creates it. (The -s option tells the archiver to list the global symbols that are defined in the library.)
- If you want to modify a library member, you can extract it, edit it, and replace it. In this example, assume there is a library named sine.obj that contains the members push.asm, pop.asm, and swap.asm.

```
ar7x -x function sine.obj
```

  The archiver makes a copy of sine.obj and places it in the current directory; it does not remove sine.obj from the library. Now you can examine or edit the extracted file. To replace the copy of sine.obj in the library with the edited copy, enter:

```
ar7x -r function sine.obj
```

- If you want to use a command file, specify the command filename after the -@ command. For example:

```
ar7x -@modules.cmd
```

  The archiver responds as follows:

```
    ==>  building archive 'modules.lib'
```

  Archiver Command File is the modules.cmd command file. The r command specifies that the filenames given in the command file replace files of the same name in the modules.lib library. The -u option specifies that

Copyright © 2026 Texas Instruments Incorporated

these files are replaced only when the current file has a more recent revision date than the file that is in the library.

Archiver Command File

```
; Command file to replace members of the
;     modules library with updated files
; Use r command and u option:
ru
; Specify library name:
modules.lib
; List filenames to be replaced if updated:
align.obj
bss.obj
data.obj
text.obj
sect.obj
clink.obj
copy.obj
double.obj
drnolist.obj
emsg.obj
end.obj
```

10.5 Library Information Archiver Description

Section 10.1 through Section 10.4 explain how to use the archiver to create libraries of object files for use in the linker of one or more applications. You can have multiple versions of the same object file libraries, each built with different sets of build options. For example, you might have different versions of your object file library for big and little endian, for different architecture revisions, or for different ABIs depending on the typical build environments of client applications. However, if you have several versions of a library, it can be cumbersome to keep track of which version of the library needs to be linked in for a particular application.

When several versions of a single library are available, the library information archiver can be used to create an index library of all of the object file library versions. This index library is used in the linker in place of a particular version of your object file library. The linker looks at the build options of the application being linked, and uses the specified index library to determine which version of your object file library to include in the linker. If one or more compatible libraries were found in the index library, the most suitable compatible library is linked in for your application.

10.5.1 Invoking the Library Information Archiver

To invoke the library information archiver, enter:

libinfo7x [*options*] **--output**=*libname libname*$_1$ [*libname*$_2$ ... *libname*$_n$]

libinfo7x	is the command that invokes the library information archiver.	
options	changes the default behavior of the library information archiver. These options are:	
	--output *libname*	specifies the name of the index library to create or update. This option is required.
	--update	updates any existing information in the index library specified with the --output option instead of creating a new index.
libnames	names individual object file libraries to be manipulated. When you enter a libname, you must enter a complete filename including extension, if applicable.	

Copyright © 2026 Texas Instruments Incorporated

10.5.2 Library Information Archiver Example

Consider these object file libraries that all have the same members, but are built with different build options:

Object File Library Name	Build Options
mylib_7100_le.lib	--silicon_version=7100
mylib_7100_be.lib	--silicon_version=7100 --big_endian

Using the library information archiver, you can create an index library called mylib.lib from the above libraries:

```
libinfo7x --output mylib.lib mylib_7100_be.lib mylib_7100_le.lib
```

You can now specify mylib.lib as a library for the linker of an application. The linker uses the index library to choose the appropriate version of the library to use. If the --issue_remarks option is specified before the --run_linker option, the linker reports which library was chosen.

- **Example 1** (little endian):

```
cl7x -mv7100 --endian=little --issue_remarks main.c -z -l lnk.cmd ./mylib.lib
<Linking>
remark: linking in "mylib_7100_le.lib" in place of "mylib.lib"
```

- **Example 2** (big endian):

```
cl7x -mv7100 --endian=big --issue_remarks main.c -z -l lnk.cmd ./mylib.lib
<Linking>
remark: linking in "mylib_7100_be.lib" in place of "mylib.lib"
```

10.5.3 Listing the Contents of an Index Library

The archiver's -t option can be used on an index library to list the archives indexed by an index library:

```
ar7x t mylib.lib
      SIZE   DATE                      FILE NAME
  --------   ------------------------  -----------------
       119   Wed Feb 03 12:45:22 2018  mylib_7100_be.lib
       119   Wed Feb 03 12:45:22 2018  mylib_7100_le.lib
         0   Wed Sep 30 12:45:22 2018  __TI_$$LIBINFO
```

The indexed object file libraries have an additional .libinfo extension in the archiver listing. The __TI_$$LIBINFO member is a special member that designates *mylib.lib* as an index library, rather than a regular library.

If the archiver's -d command is used on an index library to delete a .libinfo member, the linker will no longer choose the corresponding library when the index library is specified.

Using any other archiver option with an index library, or using -d to remove the __TI_$$LIBINFO member, results in undefined behavior, and is not supported.

10.5.4 Requirements

You must follow these requirements to use library index files:

- At least one application object file must appear on the linker command line before the index library.
- Each object file library specified as input to the library information archiver must only contain object file members that are built with the same build options.
- The linker expects the index library and all of the libraries it indexes to be in a single directory.

Copyright © 2026 Texas Instruments Incorporated

Chapter 11

Linking C/C++ Code

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

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

This chapter describes how to invoke the linker with each method. It also discusses special requirements of linking C/C++ code, including the run-time-support libraries, specifying the type of initialization, and allocating the program into memory. For a complete description of the linker, see Chapter 12 .

11.1 Invoking the Linker Through the Compiler (-z Option) 218
11.2 Linker Code Optimizations 220
11.3 Controlling the Linking Process 221

Copyright © 2026 Texas Instruments Incorporated

11.1 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.

11.1.1 Invoking the Linker Separately

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

```
cl7x --run_linker {--rom_model | --ram_model} filenames
     [options] [--output_file=name.out] --library=library [lnk.cmd]
```

cl7x --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 cl7x --run_linker without listing any C/C++ files to be compiled on the command line, you *must* use **--rom_model** or **--ram_model** on the command line or in the linker command file. The --rom_model option uses automatic variable initialization at run time; the --ram_model option uses variable initialization at load time. See Section 11.3.4 for details about using the --rom_model and --ram_model options. If you fail to specify the ROM or RAM model, you will see a linker warning that says: `warning: no suitable entry-point found; setting to 0`
filenames	Names of object files, linker command files, or archive libraries. The default extensions for input files are .c.obj (for C source files) and .cpp.obj (for C++ source files). Any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is *a.out*, unless you use the --output_file option.
options	Options affect how the linker handles your object files. Linker options can only appear after the **--run_linker** option on the command line, but otherwise may be in any order. (Options are discussed in detail in Chapter 12.)
--output_file= *name.out*	Names the output file.
--library= *library*	Identifies the appropriate archive library containing C/C++ run-time-support and floating-point math functions, or linker command files. If you are linking C/C++ code, you must use a run-time-support library. You can use the libraries included with the compiler, or you can create your own run-time-support library. If you have specified a run-time-support library in a linker command file, you do not need this parameter. The --library option's short form is -l.
lnk.cmd	Contains options, filenames, directives, or commands for the linker.

Note

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

When you specify a library as linker input, the linker includes and links only those library members that resolve undefined references. The linker uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS directives in the linker command file to customize the allocation process. For information, see Chapter 12.

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

```
cl7x --run_linker --ram_model prog1 prog2 prog3 --output_file=prog.out
     --library=rts7100_le.lib
```


Copyright © 2026 Texas Instruments Incorporated

11.1.2 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:

cl7x*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 11.1.1.

All arguments that precede --run_linker on the command line are compiler arguments. These arguments can be C/C++ source files or compiler options. These arguments are described in Section 3.2.

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

```
cl7x prog1.c prog2.c prog3.c --run_linker --ram_model --output_file=prog.out --
library=rts7100_le.lib
```

When you use cl7x --run_linker *after* listing at least one C/C++ file to be compiled on the same command line, by default the **--rom_model** is used for automatic variable initialization at run time. See Section 11.3.4 for details about using the --rom_model and --ram_model options.

Note

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

1. Object filenames from the command line
2. Arguments following the --run_linker option on the command line
3. Arguments following the --run_linker option from the C7X_C_OPTION environment variable

11.1.3 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 C7X_C_OPTION environment variable and want to selectively disable linking with the --compile_only option on the command line.

Copyright © 2026 Texas Instruments Incorporated

11.2 Linker Code Optimizations

These techniques are used to further optimize your code.

11.2.1 Conditional Linking

Conditional linking is performed by default. That is, code or data sections are not included in a link unless at least one symbol in that code or data section is referenced.

You can use the RETAIN pragma (Section 5.8.31) to force a section that contains a specific symbol to be included in the link.

You can use the CODE_SECTION (Section 5.8.5) and DATA_SECTION (Section 5.8.8) pragmas to force a symbol to be allocated in a particular section. The symbol must be referenced in a statement other than its declaration to force that section to be included in the link.

The CLINK pragma (Section 5.8.2) indicates that the section that contains the definition of this symbol is eligible for removal during conditional linking. This is the default behavior, so the CLINK pragma usually has no effect.

11.2.2 Generating Function Subsections (--gen_func_subsections Compiler Option)

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

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

Suppose a library *.c.obj file contains a signed divide routine and an unsigned divide routine. If the application requires only signed division, then only the signed divide routine is required for linking. If only one code section is used, both the signed and unsigned routines are linked in since they exist in the same *.c.obj file.

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

However, be aware that using the --gen_func_subsections compiler option can result in overall code size growth if all or nearly all functions are being referenced. This is because any section containing code must be aligned to a 64-byte boundary to support the branching mechanism. When the --gen_func_subsections option is not used, all functions in a source file are usually placed in a common section which is aligned. When --gen_func_subsections is used, each function defined in a source file is placed in a unique section. Each of the unique sections requires alignment. If all the functions in the file are required for linking, code size may increase due to the additional alignment padding for the individual subsections. Thus, the --gen_func_subsections compiler option is advantageous for use with libraries where normally only a limited number of the functions in a file are used in any one executable. The alternative to using the --gen_func_subsections option is to place each function in its own file.

The default is off if this option is not used. If this option is used without specifying "on" or "off", the default is on.

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

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

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

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

Copyright © 2026 Texas Instruments Incorporated

11.3 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 Chapter 12.

11.3.1 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.

11.3.1.1 Automatic Run-Time-Support Library Selection

The linker assumes you are using the C and C++ conventions if either the --rom_model or --ram_model linker option is specified, or if at least one C/C++ file to compile is listed on the command line. See Section 11.3.4 for details about using the --rom_model and --ram_model options.

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

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

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

Example 11-1. Using the --issue_remarks Option

```
cl7x --silicon_version=7100 --issue_remarks main.c --run_linker --rom_model
<Linking>
remark: linking in "libc.a"
remark: linking in "rts7100_le.lib" in place of "libc.a"
```

11.3.1.2 Manual Run-Time-Support Library Selection

You can bypass automatic library selection by explicitly specifying the desired run-time-support library to use. Use the --library linker option to specify the name of the library. The linker will search the path specified by the --search_path option and then the C7X_C_DIR environment variable for the named library. You can use the --library linker option on the command line or in a command file.

cl7x --run_linker {**--rom_model** | **--ram_model**} *filenames* **--library=***libraryname*

Copyright © 2026 Texas Instruments Incorporated

11.3.1.3 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.

11.3.2 Run-Time Initialization

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

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

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

Note

The _c_int00 Symbol: If you use the --ram_model or --rom_model link option, _c_int00 is automatically defined as the entry point for the program. If your command line does not list any C/C++ files to compile and does not specify either the --ram_model or --rom_model link option, the linker does not know whether or not to use the C/C++ conventions, and you will receive a linker warning that says "warning: no suitable entry-point found; setting to 0". See Section 11.3.4 for details about using the --rom_model and --ram_model options.

11.3.3 Global Object Constructors

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

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

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

Section 6.7.2.6 discusses the format of the global constructor table.

Copyright © 2026 Texas Instruments Incorporated

11.3.4 Specifying the Type of Global Variable Initialization

The C/C++ compiler produces data tables for initializing global variables. Section 6.7.2.4 discusses the format of these initialization tables. The initialization tables are used in one of the following ways:

- Global variables are initialized at *run time*. Use the --rom_model linker option (see Section 6.7.2.3).
- Global variables are initialized at *load time*. Use the --ram_model linker option (see Section 6.7.2.5).

If you use the linker command line without compiling any C/C++ files, you must use either the --rom_model or --ram_model option. These options tell the linker two things. First, they indicate that the linker should follow C/C++ conventions, using the definition of main() to link in the c_int00 boot routines. Second, they tell the linker whether to select initialization at run time or load time. If your command line fails to include one of these options when it is required, you will see "warning: no suitable entry-point found; setting to 0".

If you use a single command line to both compile and link, the --rom_model option is the default. If used, the --rom_model or --ram_model option must follow the --run_linker option (see Section 11.1).

For details on linking conventions for EABI with --rom_model and --ram_model, see Section 6.7.2.3 and Section 6.7.2.5, respectively.

11.3.5 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 11-1 summarizes the initialized sections. Table 11-2 summarizes the uninitialized sections.

Table 11-1. Initialized Sections Created by the Compiler

Name	Contents
.args	Reserved space for copying command line arguments before the main() function is called by the boot routine. See Section 3.6.
.binit	Boot time copy tables (See Section 12.8.4.2 for information on BINIT in linker command files.)
.c7xabi.exidx	Index table for exception handling; read-only (see --exceptions option).
.c7xabi.extab	Unwinding instructions for exception handling; read-only (see --exceptions option).
.cinit	The compiler does not generate a .cinit section unless the --rom_mode linker option is specified. If --rom_mode is specified, the linker creates this section, which contains tables for explicitly initialized global and static variables.
.const	Global and static const variables, including string constants and initializers for local variables.
.data	Global and static non-const variables that are explicitly initialized.
.got	Global offset table.
.init_array	Table of constructors to be called at startup.
.name.load	Compressed image of section *name*; read-only (See Section 12.8 for information on copy tables.)
.ovly	Copy tables other than boot time (.binit) copy tables. Read-only data.

Table 11-2. Uninitialized Sections Created by the Compiler

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

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

Copyright © 2026 Texas Instruments Incorporated

The linker provides MEMORY and SECTIONS directives for allocating sections. For more information about allocating sections into memory, see Section 12.5.

11.3.6 A Sample Linker Command File

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

--rom_model	Tells the linker to use autoinitialization at run time.
--heap_size	Tells the linker to set the C heap size at 0x2000 bytes.
--stack_size	Tells the linker to set the stack size to 0x0100 bytes.
--library	Tells the linker to use an archive library file, rts7100_le.lib, for input.

To link the program, use the following syntax:

cl7x --run_linker *object_file(s)* **--output_file=** *outfile* **--map_file=** *mapfile***lnk.cmd**

The MEMORY and possibly the SECTIONS directives, might require modification to work with your system. See Section 12.5 for more information on these directives.

Linker Command File

```
--rom_model
--heap_size=0x2000
--stack_size=0x0100
--library=rts7100_le.lib
MEMORY
{
    VECS:   o = 0x00000000     l = 0x000000400 /* reset & interrupt vectors     */
    PMEM:   o = 0x00000400     l = 0x00000FC00 /* intended for initialization   */
    BMEM:   o = 0x80000000     l = 0x000010000 /* .bss, .sysmem, .stack, .cinit */
}
SECTIONS
{
    vectors     >       VECS
    .text       >       PMEM
    .data       >       BMEM
    .stack      >       BMEM
    .bss        >       BMEM
    .sysmem     >       BMEM
    .cinit      >       BMEM
    .const      >       BMEM
    .cio        >       BMEM
}
```


Copyright © 2026 Texas Instruments Incorporated

Chapter 12

Linker Description

The C7000 linker creates a static executable by combining object modules. This chapter describes the linker options, directives, and statements used to create static executables. Object libraries, command files, and other key concepts are discussed as well.

The concept of sections is basic to linker operation; Chapter 8 includes a detailed discussion of sections.

12.1 Linker Overview 226
12.2 The Linker's Role in the Software Development Flow 226
12.3 Invoking the Linker 227
12.4 Linker Options 228
12.5 Linker Command Files 249
12.6 Linker Symbols 280
12.7 Default Placement Algorithm 284
12.8 Using Linker-Generated Copy Tables 285
12.9 Partial (Incremental) Linking 296
12.10 Linking C/C++ Code 296
12.11 Linker Example 298
12.12 Static Shared Objects in Multicore Applications 301

Copyright © 2026 Texas Instruments Incorporated

12.1 Linker Overview

The C7000 linker allows you to allocate output sections efficiently in the memory map. As the linker combines object files, it performs the following tasks:

- Allocates sections into the target system's configured memory
- Relocates symbols and sections to assign them to final addresses
- Resolves undefined external references between input files

The linker command language controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation. You configure system memory by defining and creating a memory model that you design. Two powerful directives, MEMORY and SECTIONS, allow you to:

- Allocate sections into specific areas of memory
- Combine object file sections
- Define or redefine global symbols at link time

12.2 The Linker's Role in the Software Development Flow

Figure 12-1 illustrates the linker's role in the software development process. The linker accepts several types of files as input, including object files, command files, libraries, and partially linked files. The linker creates an executable object module that can be downloaded to one of several development tools or executed by a C7000 device.

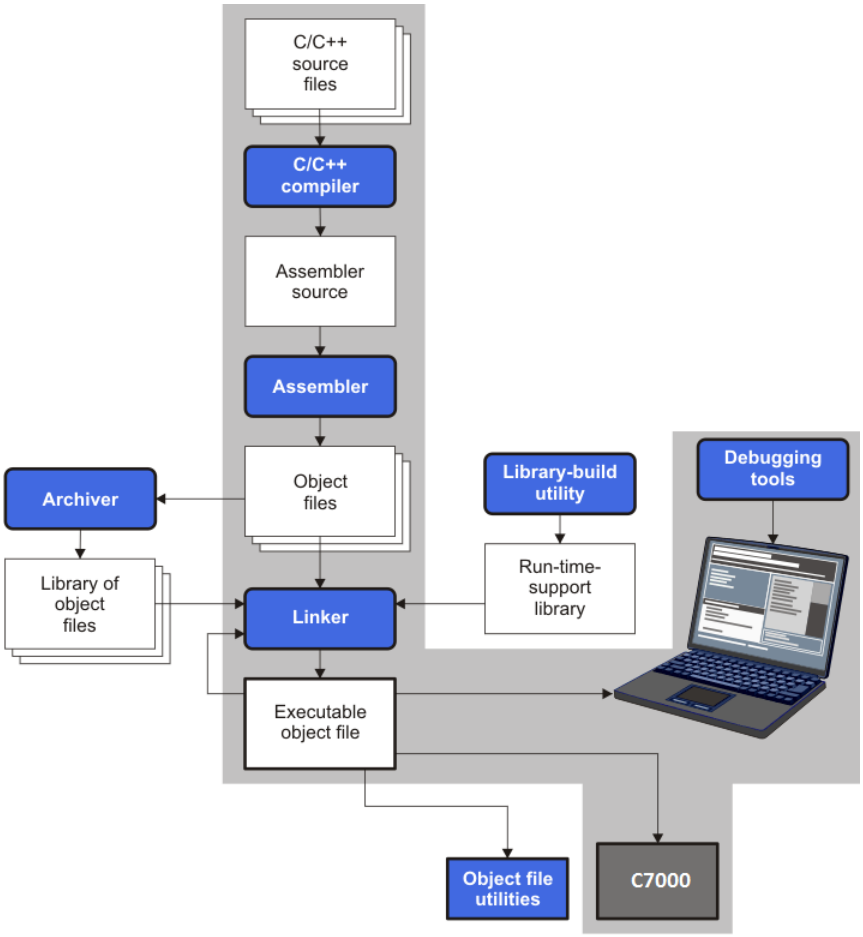


Figure 12-1. The Linker in the C7000 Software Development Flow

Copyright © 2026 Texas Instruments Incorporated

12.3 Invoking the Linker

The general syntax for invoking the linker is:

cl7x --run_linker [*options*] *filename*$_1$ .... *filename*$_n$

cl7x --run_linker	is the command that invokes the linker. The --run_linker option's short form is -z.
options	can appear anywhere on the command line or in a linker command file. (Options are discussed in Section 12.4.)
filename$_1$, *filename*$_n$	can be object files, linker command files, or archive libraries. The default extensions for input files are .c.obj (for C source files) and .cpp.obj (for C++ source files). Any other extension must be explicitly specified. The linker can determine whether the input file is an object or ASCII file that contains linker commands. The default output filename is *a.out*, unless you use the --output_file option to name the output file.

Note

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

There are two methods for invoking the linker:

- Specify options and filenames on the command line. This example links two files, file1.c.obj and file2.c.obj, and creates an output module named link.out.

```
cl7x --run_linker file1.c.obj file2.c.obj --output_file=link.out
```

- Put filenames and options in a linker command file. Filenames that are specified inside a linker command file must begin with a letter. For example, assume the file linker.cmd contains the following lines:

```
--output_file=link.out file1.c.obj file2.c.obj
```

  Now you can invoke the linker from the command line; specify the command filename as an input file:

```
cl7x --run_linker linker.cmd
```

  When you use a command file, you can also specify other options and files on the command line. For example, you could enter:

```
cl7x --run_linker --map_file=link.map linker.cmd file3.c.obj
```

  The linker reads and processes a command file as soon as it encounters the filename on the command line, so it links the files in this order: file1.c.obj, file2.c.obj, and file3.c.obj. This example creates an output file called link.out and a map file called link.map.

For information on invoking the linker for C/C++ files, see Section 12.10.

Copyright © 2026 Texas Instruments Incorporated

12.4 Linker Options

Linker options control linking operations. They can be placed on the command line or in a command file. Linker options must be preceded by a hyphen (-). Options can be separated from arguments (if they have them) by an optional space.

Table 12-1. Basic Options Summary

Option	Alias	Description	Section
--run_linker	-z	Enables linking	Section 12.3
--output_file	-o	Names the executable output module. The default filename is a.out.	Section 12.4.22
--map_file	-m	Produces a map or listing of the input and output sections, including holes, and places the listing in *filename*	Section 12.4.17
--stack_size	-stack	Sets C system stack size to *size* bytes and defines a global symbol that specifies the stack size. Default = 1K bytes	Section 12.4.27
--heap_size	-heap	Sets heap size (for the dynamic memory allocation in C) to *size* in bytes and defines a global symbol that specifies the heap size. Default = 1K bytes	Section 12.4.13

Table 12-2. File Search Path Options Summary

Option	Alias	Description	Section
--library	-l	Names an archive library or link command *filename* as linker input	Section 12.4.15
--disable_auto_rts		Disables the automatic selection of a run-time-support library	Section 12.4.8
--priority	-priority	Satisfies unresolved references by the first library that contains a definition for that symbol	Section 12.4.15.3
--reread_libs	-x	Forces rereading of libraries, which resolves back references	Section 12.4.15.3
--search_path	-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.	Section 12.4.15.1
--start-group --end-group		Create a group of libraries to be used together to resolve references.	Section 12.4.15.4

Table 12-3. Command File Preprocessing Options Summary

Option	Alias	Description	Section
--define		Predefines *name* as a preprocessor macro.	Section 12.4.10
--undefine		Removes the preprocessor macro *name*.	Section 12.4.10
--disable_pp		Disables preprocessing for command files	Section 12.4.10

Table 12-4. Diagnostic Options Summary

Option	Alias	Description	Section
--diag_error		Categorizes the diagnostic identified by *num* as an error	Section 12.4.7
--diag_remark		Categorizes the diagnostic identified by *num* as a remark	Section 12.4.7
--diag_suppress		Suppresses the diagnostic identified by *num*	Section 12.4.7
--diag_warning		Categorizes the diagnostic identified by *num* as a warning	Section 12.4.7
--display_error_number		Displays a diagnostic's identifiers along with its text	Section 12.4.7
--emit_references:file[=*file*]		Emits a file containing section information. The information includes section size, symbols defined, and references to symbols.	Section 12.4.7
--emit_warnings_as_errors	-pdew	Treats warnings as errors	Section 12.4.7
--issue_remarks		Issues remarks (nonserious warnings)	Section 12.4.7
--no_demangle		Disables demangling of symbol names in diagnostics	Section 12.4.19
--no_warnings		Suppresses warning diagnostics (errors are still issued)	Section 12.4.7
--set_error_limit		Sets the error limit to *num*. The linker abandons linking after this number of errors. (The default is 100.)	Section 12.4.7
--verbose_diagnostics		Provides verbose diagnostics that display the original source with line-wrap	Section 12.4.7
--warn_sections	-w	Displays a message when an undefined output section is created	Section 12.4.31

Copyright © 2026 Texas Instruments Incorporated

Table 12-5. Linker Output Options Summary

Option	Alias	Description	Section
--absolute_exe	-a	Produces an absolute, executable module. This is the default; if neither --absolute_exe nor --relocatable is specified, the linker acts as if --absolute_exe were specified.	Section 12.4.3.1
--mapfile_contents		Controls the information that appears in the map file.	Section 12.4.18
--relocatable	-r	Produces a nonexecutable, relocatable output module	Section 12.4.3.2
--xml_link_info		Generates a well-formed XML *file* containing detailed information about the result of a link	Section 12.4.32

Table 12-6. Symbol Management Options Summary

Option	Alias	Description	Section
--entry_point	-e	Defines a global symbol that specifies the primary entry point for the output module	Section 12.4.11
--globalize		Changes the symbol linkage to global for symbols that match *pattern*	Section 12.4.16
--hide		Hides global symbols that match *pattern*	Section 12.4.14
--localize		Changes the symbol linkage to local for symbols that match *pattern*	Section 12.4.16
--make_global	-g	Makes *symbol* global (overrides -h)	Section 12.4.16.1
--make_static	-h	Makes all global symbols static	Section 12.4.16.1
--no_sym_merge	-b	Disables merge of symbolic debugging information	Section 12.4.20
--no_symtable	-s	Strips symbol table information and line number entries from the output module	Section 12.4.21
--retain		Retains a list of sections that otherwise would be discarded.	Section 12.4.25
--scan_libraries	-scanlibs	Scans all libraries for duplicate symbol definitions	Section 12.4.26
--symbol_map		Maps symbol references to a symbol definition of a different name	Section 12.4.28
--undef_sym	-u	Places an unresolved external *symbol* into the output module's symbol table	Section 12.4.30
--unhide		Reveals (un-hides) global symbols that match *pattern*	Section 12.4.14

Table 12-7. Run-Time Environment Options Summary

Option	Alias	Description	Section
--arg_size	--args	Allocates memory to be used by the loader to pass arguments	Section 12.4.4
--fill_value	-f	Sets default fill values for holes within output sections; *fill_value* is a 64-bit constant	Section 12.4.12
--ram_model	-cr	Initializes variables at load time	Section 12.4.24
--rom_model	-c	Autoinitializes variables at run time	Section 12.4.24
--trampolines		Generates far call trampolines; on by default	Section 12.4.29

Table 12-8. Link-Time Optimization Options Summary

Option	Alias	Description	Section
--cinit_compression [=compression_kind]		Specifies the type of compression to apply to the C auto initialization data. The default *compression_kind* if this option is used with no kind specified is lzss for Lempel-Ziv-Storer-Szymanski compression. Alternately, specify --cinit_compression=rle to use Run Length Encoded compression, which generally provides less efficient compression.	Section 12.4.5
--compress_dwarf		Aggressively reduces the size of DWARF information from input object files.	Section 12.4.6
--copy_compression [=compression_kind]		Compresses data copied by linker copy tables.	Section 12.4.5
--honor_cmdfile_order		Place output sections in the order listed in the linker command file. If the linker command file specifies specific memory addresses for certain sections, those locations are honored, then the command file order is used for any remaining sections. If this option is not specified, the default is to place output sections in an order that optimizes memory usage.	--
--unused_section_elimination		Eliminates sections that are not needed in the executable module; on by default.	Section 12.4.9

Copyright © 2026 Texas Instruments Incorporated

Table 12-9. Miscellaneous Options Summary

Option	Alias	Description	Section
--linker_help	-help	Displays information about syntax and available options	–
--minimize_trampolines		Selects the trampoline minimization algorithm (argument is optional; algorithm is postorder by default)	Section 12.4.29.2
--preferred_order		Prioritizes placement of functions	Section 12.4.23
--zero_init		Controls preinitialization of uninitialized variables. Default is on. Always off if --ram_model is used.	Section 12.4.33

12.4.1 Wildcards in File, Section, and Symbol Patterns

The linker allows file, section, and symbol names to be specified using the asterisk (*) and question mark (?) wildcards. Using * matches any number of characters and using ? matches a single character. Using wildcards can make it easier to handle related objects, provided they follow a suitable naming convention. For example:

```
mp3*.obj    /* matches anything .obj that begins with mp3    */
task?.o*    /* matches task1.obj, task2.c.obj, taskX.o55, etc. */
SECTIONS
{
   .fast_code: { *.obj(*fast*) }              > FAST_MEM
   .vectors  : { vectors.c.obj(.vector:part1:*) > 0xFFFFFF00
   .str_code : { rts*.lib<str*.c.obj>(.text) }  > S1ROM
}
```

12.4.2 Specifying C/C++ Symbols with Linker Options

The link-time symbol is the same as the C/C++ identifier name. The compiler *does not* prepend an underscore to the beginning of C/C++ identifiers.

For more information on referencing symbol names, see Section 5.11.

For information specifically about C++ symbol naming, see Section 5.11.

See Section 12.6 for information about referring to linker symbols in C/C++ code.

12.4.3 Relocation Capabilities (--absolute_exe and --relocatable Options)

The linker performs relocation, which is the process of adjusting all references to a symbol when the symbol's address changes.

The linker supports two options (--absolute_exe and --relocatable) that allow you to produce an absolute or a relocatable output module. The --absolute_exe and --relocatable options may not be used together.

When the linker encounters a file that contains no relocation or symbol table information, it issues a warning message (but continues executing). Relinking an absolute file can be successful only if each input file contains no information that needs to be relocated (that is, each file has no unresolved references and is bound to the same virtual address that it was bound to when the linker created it).

12.4.3.1 Producing an Absolute Output Module (--absolute_exe option)

If you use --absolute_exe without the --relocatable option, the linker produces an *absolute, executable output module*. Absolute files contain *no* relocation information. Executable files contain the following:

- Special symbols defined by the linker (see Section 12.5.9.4)
- An header that describes information such as the program entry point
- *No* unresolved references

The following example links file1.c.obj and file2.c.obj and creates an absolute output module called a.out:

```
cl7x --run_linker --absolute_exe file1.c.obj file2.c.obj
```


Copyright © 2026 Texas Instruments Incorporated

Note

The --absolute_exe and --relocatable Options: If you do not use the --absolute_exe or the --relocatable option, the linker acts as if you specified --absolute_exe.

12.4.3.2 Producing a Relocatable Output Module (--relocatable option)

When you use the --relocatable option, the linker retains relocation entries in the output module. If the output module is relocated (at load time) or relinked (by another linker execution), use --relocatable to retain the relocation entries.

The linker produces a file that is not executable when you use the --relocatable option without the --absolute_exe option. A file that is not executable does not contain special linker symbols or an optional header. The file can contain unresolved references, but these references do not prevent creation of an output module.

This example links file1.c.obj and file2.c.obj and creates a relocatable output module called a.out:

```
cl7x --run_linker --relocatable file1.c.obj file2.c.obj
```

The output file a.out can be relinked with other object files or relocated at load time. (Linking a file that will be relinked with other files is called partial linking. For more information, see Section 12.9.)

12.4.4 Allocate Memory for Use by the Loader to Pass Arguments (--arg_size Option)

The --arg_size option instructs the linker to allocate memory to be used by the loader to pass arguments from the command line of the loader to the program. The syntax of the --arg_size option is:

--arg_size= *size*

The *size* is the number of bytes to be allocated in target memory for command-line arguments.

By default, the linker creates the __c_args__ symbol and sets it to -1. When you specify --arg_size=*size*, the following occur:

- The linker creates an uninitialized section named .args of *size* bytes.
- The __c_args__ symbol contains the address of the .args section.

The loader and the target boot code use the .args section and the __c_args__ symbol to determine whether and how to pass arguments from the host to the target program. See Section 3.6 for information about the loader.

12.4.5 Compression (--cinit_compression and --copy_compression Option)

By default, the linker does not compress copy table (Section 9.3.3 and Section 12.8) source data sections. The --cinit_compression and --copy_compression options specify compression through the linker.

The --cinit_compression option specifies the compression type the linker applies to the C autoinitialization copy table source data sections. The default is lzss.

Overlays can be managed by using linker-generated copy tables. To save ROM space the linker can compress the data copied by the copy tables. The compressed data is decompressed during copy. The --copy_compression option controls the compression of the copy data tables.

The syntax for the options are:

--cinit_compression[=*compression_kind*]

--copy_compression[=*compression_kind*]

The *compression_kind* can be one of the following types:

- **off**. Do not compress the data.
- **rle**. Compress data using Run Length Encoding.
- **lzss**. Compress data using Lempel-Ziv-Storer-Szymanski compression (default if *compression_kind* is unspecified).

Copyright © 2026 Texas Instruments Incorporated

See Section 12.8.5 for more information about compression.

12.4.6 Compress DWARF Information (--compress_dwarf Option)

The --compress_dwarf option aggressively reduces the size of DWARF information by eliminating duplicate information from input object files.

For ELF object files, which are used with EABI, the --compress_dwarf option eliminates duplicate information that could not be removed through the use of ELF COMDAT groups. (See the ELF specification for information on COMDAT groups.)

12.4.7 Control Linker Diagnostics

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

--diag_error=*num*	Categorize the diagnostic identified by *num* as an error. To find the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_error=*num* to recategorize the diagnostic as an error. You can only alter the severity of discretionary diagnostics.
--diag_remark=*num*	Categorize the diagnostic identified by *num* as a remark. To find the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_remark=*num* to recategorize the diagnostic as a remark. You can only alter the severity of discretionary diagnostics.
--diag_suppress=*num*	Suppress the diagnostic identified by *num*. To find the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_suppress=*num* to suppress the diagnostic. You can only suppress discretionary diagnostics.
--diag_warning=*num*	Categorize the diagnostic identified by *num* as a warning. To find the numeric identifier of a diagnostic message, use the --display_error_number option first in a separate link. Then use --diag_warning=*num* to recategorize the diagnostic as a warning. You can only alter the severity of discretionary diagnostics.
--display_error_number	Display a diagnostic's numeric identifier along with its text. Use this option in determining which arguments you need to supply to the diagnostic suppression options (--diag_suppress, --diag_error, --diag_remark, and --diag_warning). This option also indicates whether a diagnostic is discretionary. A discretionary diagnostic is one whose severity can be overridden. A discretionary diagnostic includes the suffix -D; otherwise, no suffix is present. See Section 3.7 for more information on understanding diagnostic messages.
--emit_references:file [=*filename*]	Emits a file containing section information. The information includes section size, symbols defined, and references to symbols. This information allows you to determine why each section is included in the linked application. The output file is a simple ASCII text file. The *filename* is used as the base name of a file created. For example, --emit_references:file=myfile generates a file named myfile.txt in the current directory.
--emit_warnings_as_errors	Treat 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	Issue remarks (nonserious warnings), which are suppressed by default.
--no_warnings	Suppress warning diagnostics (errors are still issued).
--set_error_limit=*num*	Set the error limit to *num*, which can be any decimal value. The linker abandons linking after this number of errors. (The default is 100.)
--verbose_diagnostics	Provide verbose diagnostics that display the original source with line-wrap and indicate the position of the error in the source line

12.4.8 Automatic Library Selection (--disable_auto_rts Option)

The --disable_auto_rts option disables the automatic selection of a run-time-support (RTS) library. See Section 11.3.1.1 for details on the automatic selection process.

12.4.9 Do Not Remove Unused Sections (--unused_section_elimination Option)

To minimize the footprint, the ELF linker does not include sections that are not needed to resolve any references in the final executable. Use --unused_section_elimination=off to disable this optimization. The linker default behavior is equivalent to --unused_section_elimination=on.

Copyright © 2026 Texas Instruments Incorporated

12.4.10 Linker Command File Preprocessing (--disable_pp, --define and --undefine Options)

The linker preprocesses linker command files using a standard C preprocessor. Therefore, the command files can contain well-known preprocessing directives such as #define, #include, and #if / #endif.

Three linker options control the preprocessor:

--disable_pp	Disables preprocessing for command files
--define=*name*[=*val*]	Predefines *name* as a preprocessor macro
--undefine=*name*	Removes the macro *name*

The compiler has --define and --undefine options with the same meanings. However, the linker options are distinct; only --define and --undefine options specified after --run_linker are passed to the linker. For example:

```
cl7x --define=FOO=1 main.c --run_linker --define=BAR=2 lnk.cmd
```

The linker sees only the --define for BAR; the compiler only sees the --define for FOO.

When one command file #includes another, preprocessing context is carried from parent to child in the usual way (that is, macros defined in the parent are visible in the child). However, when a command file is invoked other than through #include, either on the command line or by the typical way of being named in another command file, preprocessing context is **not** carried into the nested file. The exception to this is --define and --undefine options, which apply globally from the point they are encountered. For example:

```
--define GLOBAL
#define LOCAL
#include "incfile.cmd"     /* sees GLOBAL and LOCAL */
nestfile.cmd               /* only sees GLOBAL      */
```

Two cautions apply to the use of --define and --undefine in command files. First, they have global effect as mentioned above. Second, since they are not actually preprocessing directives themselves, they are subject to macro substitution, probably with unintended consequences. This effect can be defeated by quoting the symbol name. For example:

```
--define MYSYM=123
--undefine MYSYM     /* expands to --undefine 123 (!) */
--undefine "MYSYM"   /* ahh, that's better            */
```

The linker searches for an #include file in the following order until the file is found:

1. If the #include file name is in quotes (rather than <brackets>), search the directory that contains the current file.
2. If the --include_path compiler option was used (before the --run_linker or -z option), search the path specified with that option.
3. If the environment variable is defined, search directories pointed to by that definition. See Section 12.4.15.2.

There are two exceptions: relative pathnames (such as "../name") always search relative to the current directory, and absolute pathnames (such as "/usr/tools/name") bypass search paths entirely.

The linker provides the built-in macro definitions listed in Table 12-10. The availability of these macros within the linker is determined by the command-line options used, not the build attributes of the files being linked. If these macros are not set as expected, confirm that your project's command line uses the correct compiler option settings.

Table 12-10. Predefined C7000 Macro Names

Macro Name	Description
__DATE__	Expands to the compilation date in the form *mmm dd yyyy*
__FILE__	Expands to the current source filename

Copyright © 2026 Texas Instruments Incorporated

Table 12-10. Predefined C7000 Macro Names (continued)

Macro Name	Description
__TI_COMPILER_VERSION__	Defined to a 7-9 digit integer, depending on if X has 1, 2, or 3 digits. The number does not contain a decimal. For example, version 3.2.1 is represented as 3002001. The leading zeros are dropped to prevent the number being interpreted as an octal.
__TI_EABI__	Defined to 1 if EABI is enabled; otherwise, it is undefined.
__TIME__	Expands to the compilation time in the form "*hh:mm:ss*"
__C7000__	Always defined
__C7100__	Defined to 1 if --silicon_version=7100.
__C7120__	Defined to 1 if --silicon_version=7120.
__C7504__	Defined to 1 if --silicon_version=7504.
__C7524__	Defined to 1 if --silicon_version=7524.
__C7604__	Defined to 1 if --silicon_version=7604.

12.4.11 Define an Entry Point (--entry_point Option)

The memory address at which a program begins executing is called the *entry point*. When a loader loads a program into target memory, the program counter (PC) must be initialized to the entry point; the PC then points to the beginning of the program.

The linker can assign one of four values to the entry point. These values are listed below in the order in which the linker tries to use them. If you use one of the first three values, it must be an external symbol in the symbol table.

- The value specified by the --entry_point option. The syntax is:

  --entry_point= *global_symbol*

  where *global_symbol* defines the entry point and must be defined as an external symbol of the input files. The external symbol name of C or C++ objects may be different than the name as declared in the source language.
- The value of symbol _c_int00 (if present). The _c_int00 symbol *must* be the entry point if you are linking code produced by the C compiler.
- The value of symbol main (if present)
- 0 (default value)

This example links file1.c.obj and file2.c.obj. The symbol begin is the entry point; begin must be defined as external in file1 or file2.

```
cl7x --run_linker --entry_point=begin file1.c.obj file2.c.obj
```

See Section 12.6 for information about referring to linker symbols in C/C++ code.

12.4.12 Set Default Fill Value (--fill_value Option)

The --fill_value option fills the holes formed within output sections. The syntax for the option is:

--fill_value= *value*

The argument *value* is a 64-bit constant (up to 16 hexadecimal digits). If you do not use --fill_value, the linker uses 0 as the default fill value.

This example fills holes with the hexadecimal value ABCDABCD:

```
cl7x --run_linker --fill_value=0xABCDABCDABCDABCD file1.c.obj file2.c.obj
```


Copyright © 2026 Texas Instruments Incorporated

12.4.13 Define Heap Size (--heap_size Option)

The C/C++ compiler uses an uninitialized section called for the C run-time memory pool used by malloc(). You can set the size of this memory pool at link time by using the --heap_size option. The syntax for the --heap_size option is:

--heap_size= *size*

The *size* must be a constant. This example defines a 4K byte heap:

```
cl7x --run_linker --heap_size=0x1000 /* defines a 4k heap (.sysmem section)*/
```

The linker creates the section only if there is a section in an input file.

The linker also creates a global symbol, , and assigns it a value equal to the size of the heap. The default size is 1K bytes. See Section 12.6 for information about referring to linker symbols in C/C++ code.

12.4.14 Hiding Symbols

Symbol hiding prevents the symbol from being listed in the output file's symbol table. While localization is used to prevent name space clashes in a link unit (see Section 12.4.16), symbol hiding is used to obscure symbols which should not be visible outside a link unit. Such symbol's names appear only as empty strings or "no name" in object file readers. The linker supports symbol hiding through the --hide and --unhide options.

The syntax for these options are:

--hide='*pattern***'**

--unhide='*pattern***'**

The *pattern* is a "glob" (a string with optional ? or * wildcards). Use ? to match a single character. Use * to match zero or more characters.

The --hide option hides global symbols with a linkname matching the *pattern*. It hides symbols matching the pattern by changing the name to an empty string. A global symbol that is hidden is also localized.

The --unhide option reveals (un-hides) global symbols that match the *pattern* that are hidden by the --hide option. The --unhide option excludes symbols that match pattern from symbol hiding provided the pattern defined by --unhide is more restrictive than the pattern defined by --hide.

These options have the following properties:

- The --hide and --unhide options can be specified more than once on the command line.
- The order of --hide and --unhide has no significance.
- A symbol is matched by only one pattern defined by either --hide or --unhide.
- A symbol is matched by the most restrictive pattern. Pattern A is considered more restrictive than Pattern B, if Pattern A matches a narrower set than Pattern B.
- It is an error if a symbol matches patterns from --hide and --unhide and one does not supersede the other. Pattern A supersedes pattern B if A can match everything B can and more. If Pattern A supersedes Pattern B, then Pattern B is said to more restrictive than Pattern A.
- These options affect final and partial linking.

In map files these symbols are listed under the Hidden Symbols heading.

12.4.15 Alter the Library Search Algorithm (--library, --search_path, and C7X_C_DIR)

Usually, when you want to specify a file as linker input, you simply enter the filename; the linker looks for the file in the current directory. For example, suppose the current directory contains the library object.lib. If this library defines symbols that are referenced in the file file1.c.obj, this is how you link the files:

```
cl7x --run_linker file1.c.obj object.lib
```


Copyright © 2026 Texas Instruments Incorporated

To use a file that is not in the current directory, use the --library linker option. The --library option's short form is -l. The syntax for this option is:

--library=[*pathname*] *filename*

The *filename* is the name of an archive, object file, or linker command file. Specify up to 128 search paths.

The --library option is not required when one or more members of an object library are specified for input to an output section. For more information about allocating archive members, see Section 12.5.5.5.

You can adjust the linker's directory search algorithm using the --search_path linker option or the C7X_C_DIR environment variable. The linker searches for object libraries and command files in this order:

1. Search directories named with the --search_path linker option. The --search_path option must appear before the --library option on the command line or in a command file.
2. Search directories named with C7X_C_DIR.
3. If C7X_C_DIR is not set, search directories named with the C7X_A_DIR environment variable.
4. Search the current directory.

12.4.15.1 Name an Alternate Library Directory (--search_path Option)

The --search_path option names an alternate directory that contains input files. The --search_path option's short form is `-I`. The syntax for this option is:

--search_path= *pathname*

The *pathname* names a directory that contains input files.

When the linker searches for files named with the --library option, it searches through directories named with --search_path first. Each --search_path option specifies only one directory, but you can use several --search_path options per invocation. If you use the --search_path option to name an alternate directory, it must precede any --library option on the command line or in a command file.

For example, assume that there are two archive libraries called r.lib and lib2.lib that reside in ld and ld2 directories. The table below shows the directories that r.lib and lib2.lib reside in, how to set environment variable, and how to use both libraries during a link. Select the row for your operating system:

Operating System	Enter
UNIX (Bourne shell)	`cl7x --run_linker f1.c.obj f2.c.obj --search_path=/ld --search_path=/ld2 --library=r.lib --library=lib2.lib`
Windows	`cl7x --run_linker f1.c.obj f2.c.obj --search_path=\ld --search_path=\ld2 --library=r.lib --library=lib2.lib`

12.4.15.2 Name an Alternate Library Directory (C7X_C_DIR Environment Variable)

An environment variable is a system symbol that you define and assign a string to. The linker uses an environment variable named C7X_C_DIR to name alternate directories that contain object libraries. The command syntaxes for assigning the environment variable are:

Operating System	Enter
UNIX (Bourne shell)	**C7X_C_DIR ="***pathname*$_1$**;** *pathname*$_2$**; . . . "; export C7X_C_DIR**
Windows	**set C7X_C_DIR =** *pathname*$_1$ **;** *pathname*$_2$ **; . . .**

The *pathnames* are directories that contain input files. Use the --library linker option on the command line or in a command file to tell the linker which library or linker command file to search for. The pathnames must follow these constraints:

- Pathnames must be separated with a semicolon.

Copyright © 2026 Texas Instruments Incorporated

- 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 C7X_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 C7X_C_DIR=c:\first path\to\tools;d:\second path\to\tools
```

In the example below, assume that two archive libraries called r.lib and lib2.lib reside in ld and ld2 directories. The table below shows how to set the environment variable, and how to use both libraries during a link. Select the row for your operating system:

Operating System	Invocation Command
UNIX (Bourne shell)	`C7X_C_DIR="/ld ;/ld2"; export C7X_C_DIR;` `cl7x --run_linker f1.c.obj f2.c.obj --library=r.lib --library=lib2.lib`
Windows	`C7X_C_DIR=\ld;\ld2` `cl7x --run_linker f1.c.obj f2.c.obj --library=r.lib --library=lib2.lib`

The environment variable remains set until you reboot the system or reset the variable by entering:

Operating System	Enter
UNIX (Bourne shell)	`unset C7X_C_DIR`
Windows	`set C7X_C_DIR=`

The assembler uses an environment variable named C7X_A_DIR to name alternate directories that contain copy/include files. If C7X_C_DIR is not set, the linker searches for object libraries in the directories named with C7X_A_DIR. For more information about object libraries, see Section 12.6.7.

12.4.15.3 Exhaustively Read and Search Libraries (--reread_libs and --priority Options)

There are two ways to exhaustively search for unresolved symbols:

- Reread libraries if you cannot resolve a symbol reference (--reread_libs).
- Search libraries in the order that they are specified (--priority).

The linker normally reads input files, including archive libraries, only once when they are encountered on the command line or in the command file. When an archive is read, any members that resolve references to undefined symbols are included in the link. If an input file later references a symbol defined in a previously read archive library, the reference is not resolved.

With the --reread_libs option, you can force the linker to reread all libraries. The linker rereads libraries until no more references can be resolved. Linking using --reread_libs may be slower, so you should use it only as needed. For example, if a.lib contains a reference to a symbol defined in b.lib, and b.lib contains a reference to a symbol defined in a.lib, you can resolve the mutual dependencies by listing one of the libraries twice, as in:

```
cl7x --run_linker --library=a.lib --library=b.lib --library=a.lib
```

or you can force the linker to do it for you:

```
cl7x --run_linker --reread_libs --library=a.lib --library=b.lib
```


Copyright © 2026 Texas Instruments Incorporated

The --priority option provides an alternate search mechanism for libraries. Using --priority causes each unresolved reference to be satisfied by the first library that contains a definition for that symbol. For example:

```
objfile  references A
lib1     defines B
lib2     defines A, B; obj defining A references B
```

Under the existing model, objfile resolves its reference to A in lib2, pulling in a reference to B, which resolves to the B in lib2.

Under --priority, objfile resolves its reference to A in lib2, pulling in a reference to B, but now B is resolved by searching the libraries in order and resolves B to the first definition it finds, namely the one in lib1.

The --priority option is useful for libraries that provide overriding definitions for related sets of functions in other libraries without having to provide a complete version of the whole library. For example, suppose you want to override versions of malloc and free defined in the rts7100_le.lib without providing a full replacement for rts7100_le.lib. Using --priority and linking your new library before rts7100_le.lib guarantees that all references to malloc and free resolve to the new library.

The --priority option is intended to support linking RTOS applications where situations like the one illustrated above occur.

12.4.15.4 Specify Related Libraries (--start-group and --end-group)

The --start-group and --end-group options can be used to associate a set of library files. These files will be treated as a single library, which will be iteratively scanned in the order listed until no more undefined symbols can be resolved by the group. These options are useful for resolving backwards or cyclical dependencies across libraries without listing the libraries on the command line multiple times.

These grouping options are similar to the --reread_libs (-x) linker option, except that the group options provide finer control over which libraries to iterate over. The grouping options can be used with the --reread_libs option.

These grouping options are not compatible with the --priority option. If the --priority option is used, the libraries specified with the grouping options are ignored.

For example:

```
cl7x --run_linker --start-group a.lib b.lib --end-group c.lib
```

Suppose the definitions and functions used in this example are as follows, where there is a cyclic dependency between a.lib and b.lib and there is an alternate definition of fcn_a() in c.lib:

- main.c.obj: defines main(), which uses fcn()
- a.lib: defines fcn_a(), which uses fcn_b()
- b.lib: defines fcn(), which uses fcn_a(); defines fcn_b()
- c.lib: defines fcn_a()

The linker scans the files and the following occurs:

1. Scans main.c.obj. Defines main() and adds fcn() to the undefined list.
2. Encounters the --start-group
3. Scans a.lib. No definitions in a.lib are undefined.
4. Scans b.lib. Removes fcn() from the undefined list. Adds fcn_a() to the undefined list.
5. End of group is encountered and undefines have been added, so scan group again.
6. Scan of a.lib defines fcn_a() and removes it from the undefined list. Add fcn_b() to the undefined list.
7. Scan of b.lib defines fcn_b() and removes it from the undefined list.
8. Undefines have been added, so rescan the group.
9. Since no definitions in a.lib or b.lib are undefined, exit the group.
10. No definitions in c.lib are undefined.
11. Cyclical dependency between a.lib and b.lib is resolved before encountering the alternate definition in c.lib.

Copyright © 2026 Texas Instruments Incorporated

12.4.16 Change Symbol Localization

Symbol localization changes symbol linkage from global to local (static). This is used to obscure global symbols that should not be widely visible, but must be global because they are accessed by several modules in the library. The linker supports symbol localization through the --localize and --globalize linker options.

The syntax for these options are:

--localize='*pattern***'**

--globalize='*pattern***'**

The *pattern* is a "glob" (a string with optional ? or * wildcards). Use ? to match a single character. Use * to match zero or more characters.

The --localize option changes the symbol linkage to local for symbols matching the *pattern*.

The --globalize option changes the symbol linkage to global for symbols matching the *pattern*. The --globalize option only affects symbols that are localized by the --localize option. The --globalize option excludes symbols that match the pattern from symbol localization, provided the pattern defined by --globalize is more restrictive than the pattern defined by --localize.

See Section 12.4.2 for information about using C/C++ identifiers in linker options such as --localize and --globalize.

These options have the following properties:

- The --localize and --globalize options can be specified more than once on the command line.
- The order of --localize and --globalize options has no significance.
- A symbol is matched by only one pattern defined by either --localize or --globalize.
- A symbol is matched by the most restrictive pattern. Pattern A is considered more restrictive than Pattern B, if Pattern A matches a narrower set than Pattern B.
- It is an error if a symbol matches patterns from --localize and --globalize and if one does not supersede other. Pattern A supersedes pattern B if A can match everything B can, and some more. If Pattern A supersedes Pattern B, then Pattern B is said to more restrictive than Pattern A.
- These options affect final and partial linking.

In map files these symbols are listed under the Localized Symbols heading.

12.4.16.1 Make All Global Symbols Static (--make_static Option)

The --make_static option makes all global symbols static. Static symbols are not visible to externally linked modules. By making global symbols static, global symbols are essentially hidden. This allows external symbols with the same name (in different files) to be treated as unique.

The --make_static option effectively nullifies all .global assembler directives. All symbols become local to the module in which they are defined, so no external references are possible. For example, assume file1.c.obj and file2.c.obj both define global symbols called EXT. By using the --make_static option, you can link these files without conflict. The symbol EXT defined in file1.c.obj is treated separately from the symbol EXT defined in file2.c.obj.

```
cl7x --run_linker --make_static file1.c.obj file2.c.obj
```

The --make_static option makes all global symbols static. If you have a symbol that you want to remain global and you use the --make_static option, you can use the --make_global option to declare that symbol to be global. The --make_global option overrides the effect of the --make_static option for the symbol that you specify. The syntax for the --make_global option is:

--make_global= *global_symbol*

Copyright © 2026 Texas Instruments Incorporated

12.4.17 Create a Map File (--map_file Option)

The syntax for the --map_file option is:

--map_file= *filename*

The linker map describes:

- Memory configuration
- Input and output section allocation
- Linker-generated copy tables
- Trampolines
- The addresses of external symbols after they have been relocated
- Hidden and localized symbols

The map file contains the name of the output module and the entry point; it can also contain up to three tables:

- A table shows the new memory configuration if the MEMORY directive specifies any non-default configuration. The table has the following columns, which are generated from the MEMORY directive in the linker command file. For information about the MEMORY directive, see Section 12.5.4.
  - **Name.** This is the name of the memory range specified with the MEMORY directive.
  - **Origin.** This specifies the starting address of a memory range.
  - **Length.** This specifies the length of a memory range.
  - **Unused.** This specifies the total amount of unused (available) memory in that memory area.
  - **Attributes.** This specifies one to four attributes associated with the named range:

R	specifies that the memory can be read.
W	specifies that the memory can be written to.
X	specifies that the memory can contain executable code.
I	specifies that the memory can be initialized.

- A table showing the linked addresses of each output section and the input sections that make up the output sections (section placement map). This table has the following columns; this information is generated on the basis of the information in the SECTIONS directive in the linker command file:
  - **Output section.** This is the name of the output section specified with the SECTIONS directive.
  - **Origin.** The first origin listed for each output section is the starting address of that output section. The indented origin value is the starting address of that portion of the output section.
  - **Length.** The first length listed for each output section is the length of that output section. The indented length value is the length of that portion of the output section.
  - **Attributes/input sections.** This lists the input file or value associated with an output section. If the input section could not be allocated, the map file will indicate this with "FAILED TO ALLOCATE".

  For more information about the SECTIONS directive, see Section 12.5.5.
- A table showing each external symbol and its address sorted by symbol name.
- A table showing each external symbol and its address sorted by symbol address.

The following example links file1.c.obj and file2.c.obj and creates a map file called map.out:

```
cl7x --run_linker file1.c.obj file2.c.obj --map_file=map.out
```

Output Map File, demo.map shows an example of a map file.

Copyright © 2026 Texas Instruments Incorporated

12.4.18 Manage Map File Contents (--mapfile_contents Option)

The --mapfile_contents option assists with managing the content of linker-generated map files. The syntax for the --mapfile_contents option is:

--mapfile_contents= *filter*[, *filter*]

When the --map_file option is specified, the linker produces a map file containing information about memory usage, placement information about sections that were created during a link, details about linker-generated copy tables, and symbol values.

The --mapfile_contents option provides a mechanism for you to control what information is included in or excluded from a map file. When you specify --mapfile_contents=help from the command line, a help screen listing available filter options is displayed. The following filter options are available:

Attribute	Description	Default State
copytables	Copy tables	On
entry	Entry point	On
load_addr	Display load addresses	Off
memory	Memory ranges	On
modules	Module view	On
sections	Sections	On
sym_defs	Defined symbols per file	Off
sym_dp	Symbols sorted by data page	On
sym_name	Symbols sorted by name	On
sym_runaddr	Symbols sorted by run address	On
all	Enables all attributes	
none	Disables all attributes	

The --mapfile_contents option controls display filter settings by specifying a comma-delimited list of display attributes. When prefixed with the word no, an attribute is disabled instead of enabled. For example:

```
--mapfile_contents=copytables,noentry
--mapfile_contents=all,nocopytables
--mapfile_contents=none,entry
```

By default, those sections that are currently included in the map file when the --map_file option is specified are included. The filters specified in the --mapfile_contents options are processed in the order that they appear in the command line. In the third example above, the first filter, none, clears all map file content. The second filter, entry, then enables information about entry points to be included in the generated map file. That is, when --mapfile_contents=none,entry is specified, the map file contains *only* information about entry points.

The load_addr and sym_defs attributes are both disabled by default.

If you turn on the load_addr filter, the map file includes the load address of symbols that are included in the symbol list in addition to the run address (if the load address is different from the run address).

You can use the sym_defs filter to include information sorted on a file by file basis. You may find it useful to replace the sym_name, sym_dp, and sym_runaddr sections of the map file with the sym_defs section by specifying the following --mapfile_contents option:

```
--mapfile_contents=nosym_name,nosym_dp,nosym_runaddr,sym_defs
```

By default, information about global symbols defined in an application are included in tables sorted by name, data page, and run address. If you use the --mapfile_contents=sym_defs option, static variables are also listed.

Copyright © 2026 Texas Instruments Incorporated

12.4.19 Disable Name Demangling (--no_demangle)

By default, the linker uses demangled symbol names in diagnostics. For example:

```
undefined symbol                first referenced in file
ANewClass::getValue()           test.cpp.obj
```

The --no_demangle option instead shows the linkname for symbols in diagnostics. For example:

```
undefined symbol                first referenced in file
_ZN9ANewClass8getValueEv        test.cpp.obj
```

For information on referencing object file symbol names, see Section 5.11. For information specifically about linker symbol naming, see Section 12.6.

12.4.20 Merging of Symbolic Debugging Information

By default, the linker eliminates duplicate entries of symbolic debugging information. Such duplicate information is commonly generated when a C program is compiled for debugging. For example:

```
-[ header.h ]-
typedef struct
{
   <define some structure members>
} XYZ;
-[ f1.c ]-
#include "header.h"
...
-[ f2.c ]-
#include "header.h"
...
```

When these files are compiled for debugging, both f1.c.obj and f2.c.obj have symbolic debugging entries to describe type XYZ. For the final output file, only one set of these entries is necessary. The linker eliminates the duplicate entries automatically.

Use the --no_sym_merge option if you want the linker to keep such duplicate entries in object files. Using the --no_sym_merge option has the effect of the linker running faster and using less host memory during linking, but the resulting executable file may be very large due to duplicated debug information.

12.4.21 Strip Symbolic Information (--no_symtable Option)

The --no_symtable option creates a smaller output module by omitting symbol table information and line number entries. The --no_sym_table option is useful for production applications when you do not want to disclose symbolic information to the consumer.

This example links file1.c.obj and file2.c.obj and creates an output module, stripped of line numbers and symbol table information, named nosym.out:

```
cl7x --run_linker --output_file=nosym.out --no_symtable file1.c.obj file2.c.obj
```

Using the --no_symtable option limits later use of a symbolic debugger.

Note

Stripping Symbolic Information: The --no_symtable option is deprecated. To remove symbol table information, use the strip7x utility as described in Section 13.4.

Copyright © 2026 Texas Instruments Incorporated

12.4.22 Name an Output Module (--output_file Option)

The linker creates an output module when no errors are encountered. If you do not specify a filename for the output module, the linker gives it the default name a.out. If you want to write the output module to a different file, use the --output_file option. The syntax for the --output_file option is:

--output_file= *filename*

The *filename* is the new output module name.

This example links file1.c.obj and file2.c.obj and creates an output module named run.out:

```
cl7x --run_linker --output_file=run.out file1.c.obj file2.c.obj
```

12.4.23 Prioritizing Function Placement (--preferred_order Option)

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

--preferred_order=*function specification*

12.4.24 C Language Options (--ram_model and --rom_model Options)

The --ram_model and --rom_model options cause the linker to use linking conventions that are required by the C compiler. Both options inform the linker that the program is a C program and requires a boot routine.

- The --ram_model option tells the linker to initialize variables at load time.
- The --rom_model option tells the linker to autoinitialize variables at run time.

If you use a linker command line that does not compile any C/C++ files, you must use either the --rom_model or --ram_model option. If your command line fails to include one of these options when it is required, you will see "warning: no suitable entry-point found; setting to 0".

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

For more information, see Section 12.10, Section 9.3.2.1, and Section 9.3.2.2.

12.4.25 Retain Discarded Sections (--retain Option)

When --unused_section_elimination is on, the ELF linker does not include a section in the final link if it is not needed in the executable to resolve references. The --retain option tells the linker to retain a list of sections that would otherwise not be retained. This option accepts the wildcards '*' and '?'. When wildcards are used, the argument should be in quotes. The syntax for this option is:

--retain=*sym_or_scn_spec*

The --retain option take one of the following forms:

- **--retain=** *symbol_spec*

  Specifying the symbol format retains sections that define *symbol_spec*. For example, this code retains sections that define symbols that start with init:

  ```
  --retain='init*'
  ```

  You cannot specify --retain='*'.

Copyright © 2026 Texas Instruments Incorporated

- **--retain=** *file_spec*(*scn_spec*[, *scn_spec*, ...]

  Specifying the file format retains sections that match one or more *scn_spec* from files matching the *file_spec*. For example, this code retains .intvec sections from all input files:

  ```
  --retain='*(.int*)'
  ```

  You can specify --retain='*(*)' to retain all sections from all input files. However, this does not prevent sections from library members from being optimized out.
- **--retain=***ar_spec*<*mem_spec*, [*mem_spec*, ...>(*scn_spec*[, *scn_spec*, ...]

  Specifying the archive format retains sections matching one or more *scn_spec* from members matching one or more *mem_spec* from archive files matching *ar_spec*. For example, this code retains the .text sections from printf.c.obj in the rts7100_le.lib library:

  ```
  --retain=rts7100_le.lib<printf.c.obj>(.text)
  ```

  If the library is specified with the --library option (--library=rts7100_le.lib) the library search path is used to search for the library. You cannot specify '*<*>(*)'.

12.4.26 Scan All Libraries for Duplicate Symbol Definitions (--scan_libraries)

The --scan_libraries option scans all libraries during a link looking for duplicate symbol definitions to those symbols that are actually included in the link. The scan does not consider absolute symbols or symbols defined in COMDAT sections. The --scan_libraries option helps determine those symbols that were actually chosen by the linker over other existing definitions of the same symbol in a library.

The library scanning feature can be used to check against unintended resolution of a symbol reference to a definition when multiple definitions are available in the libraries.

12.4.27 Define Stack Size (--stack_size Option)

The C7000 C/C++ compiler uses an uninitialized section, .stack, to allocate space for the run-time stack. You can set the size of this section in bytes at link time with the --stack_size option. The syntax for the --stack_size option is:

--stack_size= *size*

The *size* must be a constant and is in bytes. This example defines a 4K bytes stack:

```
cl7x --run_linker --stack_size=0x1000 /* defines a 4K heap (.stack section)*/
```

If you specified a different stack size in an input section, the input section stack size is ignored. Any symbols defined in the input section remain valid; only the stack size is different.

When the linker defines the .stack section, it also defines a global symbol, , and assigns it a value equal to the size of the section. The default software stack size is 1K bytes. See Section 12.6 for information about referring to linker symbols in C/C++ code.

12.4.28 Mapping of Symbols (--symbol_map Option)

Symbol mapping allows a symbol reference to be resolved by a symbol with a different name, which allows functions to be overridden with alternate definitions. This can be used to patch in alternate implementations to provide patches (bug fixes) or alternate functionality. The syntax for the --symbol_map option is:

--symbol_map= *refname=defname*

For example, the following code makes the linker resolve any references to foo by the definition foo_patch:

```
--symbol_map=foo=foo_patch
```


Copyright © 2026 Texas Instruments Incorporated

The string passed with the --symbol_map option should contain no spaces and not be surrounded by quotes. This allows the same linker option syntax to work on the command line, in a linker command file, and in an options file.

12.4.29 Generate Far Call Trampolines (--trampolines Option)

The C7000 device has PC-relative call and PC-relative branch instructions whose range is smaller than the entire address space. When these instructions are used, the destination address must be near enough to the instruction that the difference between the call and the destination fits in the available encoding bits. If the called function is too far away from the calling function, the linker generates an error or generates a trampoline, depending on the setting of the --trampolines option (on or off).

The alternative to a PC-relative call is an absolute call, which is often implemented as an indirect call: load the called address into a register, and call that register. This is often undesirable because it takes more instructions (speed- and size-wise) and requires an extra register to contain the address.

By default, the compiler generates calls that may require a trampoline if the destination is too far away. On some architectures, this type of call is called a "near call."

The --trampolines option allows you to control the generation of trampolines. When set to "on", this option causes the linker to generate a trampoline code section for each call that is linked out-of-range of its called destination. The trampoline code section contains a sequence of instructions that performs a transparent long branch to the original called address. Each calling instruction that is out-of-range from the called function is redirected to the trampoline.

The syntax for this option is:

--trampolines[=on|off]

The default setting is on. For C7000, trampolines are turned on by default.

For example, in a section of C code the bar function calls the foo function. The compiler generates this code for the function:

```
bar:
      ...
      CALL .B1    foo     ; call the function "foo"
      ...
```

If the foo function is placed out-of-range from the call to foo that is inside of bar, then with --trampolines the linker changes the original call to foo into a call to foo_trampoline as shown:

```
bar:
      ...
      CALL .B1 $Tramp$L$PI$$myfunc ; call a trampoline for foo
      ...
```

The above code generates a trampoline code section called foo_trampoline, which contains code that executes a long branch to the original called function, foo. For example:

```
$Tramp$L$PI$$myfunc:
    BE .B1 foo ; long branch to foo (with constant extension)
```

Trampolines can be shared among calls to the same called function. The only requirement is that all calls to the called function be linked near the called function's trampoline.

When the linker produces a map file (the --map_file option) and it has produced one or more trampolines, then the map file will contain statistics about what trampolines were generated to reach which functions. A list of calls for each trampoline is also provided in the map file.

Copyright © 2026 Texas Instruments Incorporated

Note

The Linker Assumes D15 Contains the Stack Pointer

Assembly language programmers must be aware that the linker assumes D15 contains the stack pointer. The linker must save and restore values on the stack in trampoline code that it generates. If you do not use D15 as the stack pointer, you should use the linker option that disables trampolines, --trampolines=off. Otherwise, trampolines could corrupt memory and overwrite register values.

12.4.29.1 Advantages and Disadvantages of Using Trampolines

The advantage of using trampolines is that you can treat all calls as near calls, which are faster and more efficient. You will only need to modify those calls that do not reach. In addition, there is little need to consider the relative placement of functions that call each other. Cases where calls must go through a trampoline are less common than near calls.

While generating far call trampolines provides a more straightforward solution, trampolines have the disadvantage that they are somewhat slower than directly calling a function. They require both a call and a branch. Additionally, while inline code could be tailored to the environment of the call, trampolines are generated in a more general manner, and may be slightly less efficient than inline code.

An alternative method to creating a trampoline code section for a call that cannot reach its called function is to actually modify the source code for the call. In some cases this can be done without affecting the size of the code. However, in general, this approach is extremely difficult, especially when the size of the code is affected by the transformation.

12.4.29.2 Minimizing the Number of Trampolines Required (--minimize_trampolines Option)

The --minimize_trampolines option attempts to place sections so as to minimize the number of far call trampolines required, possibly at the expense of optimal memory packing. The syntax is:

--minimize_trampolines=postorder

The argument selects a heuristic to use. The postorder heuristic attempts to place functions before their callers, so that the PC-relative offset to the callee is known when the caller is placed. By placing the callee first, its address is known when the caller is placed so the linker can definitively know if a trampoline is required.

12.4.29.3 Carrying Trampolines From Load Space to Run Space

It is sometimes useful to load code in one location in memory and run it in another. The linker provides the capability to specify separate load and run allocations for a section. The burden of actually copying the code from the load space to the run space is left to you.

A copy function must be executed before the real function can be executed in its run space. To facilitate this copy function, the assembler provides the .label directive, which allows you to define a load-time address. These load-time addresses can then be used to determine the start address and size of the code to be copied. However, this mechanism will *not* work if the code contains a call that requires a trampoline to reach its called function. This is because the trampoline code is generated at link time, after the load-time addresses associated with the .label directive have been defined. If the linker detects the definition of a .label symbol in an input section that contains a trampoline call, then a warning is generated.

To solve this problem, you can use the START(), END(), and SIZE() operators (see Section 12.5.9.7). These operators allow you to define symbols to represent the load-time start address and size inside the linker command file. These symbols can be referenced by the copy code, and their values are not resolved until link time, after the trampoline sections have been allocated.

Copyright © 2026 Texas Instruments Incorporated

Here is an example of how you could use the START() and SIZE() operators in association with an output section to copy the trampoline code section along with the code containing the calls that need trampolines:

```
SECTIONS
{  .foo : load = ROM, run = RAM, start(foo_start), size(foo_size)
          { x.obj(.text) }
   .text: {} > ROM
   .far : { --library=rts.lib(.text) } > FAR_MEM
}
```

A function in x.c.obj contains an run-time-support call. The run-time-support library is placed in far memory and so the call is out-of-range. A trampoline section will be added to the .foo output section by the linker. The copy code can refer to the symbols foo_start and foo_size as parameters for the load start address and size of the entire .foo output section. This allows the copy code to copy the trampoline section along with the original x.c.obj code in .text from its load space to its run space.

See Section 12.6 for information about referring to linker symbols in C/C++ code.

12.4.30 Introduce an Unresolved Symbol (--undef_sym Option)

The --undef_sym option introduces the linkname for an unresolved symbol into the linker's symbol table. This forces the linker to search a library and include the member that defines the symbol. The linker must encounter the --undef_sym option *before* it links in the member that defines the symbol. The syntax for the --undef_sym option is:

--undef_sym= *symbol*

For example, suppose a library named rts7100_le.lib contains a member that defines the symbol symtab; none of the object files being linked reference symtab. However, suppose you plan to relink the output module and you want to include the library member that defines symtab in this link. Using the --undef_sym option as shown below forces the linker to search rts7100_le.lib for the member that defines symtab and to link in the member.

```
cl7x --run_linker --undef_sym=symtab file1.c.obj file2.c.obj rts7100_le.lib
```

If you do not use --undef_sym, this member is not included, because there is no explicit reference to it in file1.c.obj or file2.c.obj.

12.4.31 Display a Message When an Undefined Output Section Is Created (--warn_sections)

In a linker command file, you can set up a SECTIONS directive that describes how input sections are combined into output sections. However, if the linker encounters one or more input sections that do not have a corresponding output section defined in the SECTIONS directive, the linker combines input sections that have the same name into an output section with that name. By default, the linker does not display a message to tell you that this occurred.

Use the --warn_sections option to cause the linker to display a message when it creates a new output section.

For more information about the SECTIONS directive, see Section 12.5.5. For more information about the default actions of the linker, see Section 12.7.

12.4.32 Generate XML Link Information File (--xml_link_info Option)

The linker supports the generation of an XML link information file through the --xml_link_info=*file* option. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker generated map file. See Appendix A for specifics on the contents of the generated XML file.

Copyright © 2026 Texas Instruments Incorporated

12.4.33 Zero Initialization (--zero_init Option)

The C and C++ standards require that global and static variables that are not explicitly initialized must be set to 0 before program execution. The C/C++ compiler supports preinitialization of uninitialized variables by default. To turn this off, specify the linker option --zero_init=off.

The syntax for the --zero_init option is:

--zero_init[={*on*|*off*}]

Zero initialization takes place only if the --rom_model linker option, which causes autoinitialization to occur, is used. If you use the --ram_model option for linking, the linker does not generate initialization records, and the loader must handle both data and zero initialization.

Note

Disabling Zero Initialization Not Recommended: In general, disabling zero initialization is not recommended. If you turn off zero initialization, automatic initialization of uninitialized global and static objects to zero will not occur. You are then expected to initialize these variables to zero in some other manner.

Copyright © 2026 Texas Instruments Incorporated

12.5 Linker Command Files

Linker command files allow you to put linker options and directives in a file; this is useful when you invoke the linker often with the same options and directives. Linker command files are also useful because they allow you to use the MEMORY and SECTIONS directives to customize your application. You must use these directives in a command file; you cannot use them on the command line.

Linker command files are ASCII files that contain one or more of the following:

- Input filenames, which specify object files, archive libraries, or other command files. (If a command file calls another command file as input, this statement must be the *last* statement in the calling command file. The linker does not return from called command files.)
- Linker options, which can be used in the command file in the same manner that they are used on the command line
- The MEMORY and SECTIONS linker directives. The MEMORY directive defines the target memory configuration (see Section 12.5.4). The SECTIONS directive controls how sections are built and allocated (see Section 12.5.5.)
- Assignment statements, which define and assign values to global symbols

To invoke the linker with a command file, enter the cl7x --run_linker command and follow it with the name of the command file:

cl7x --run_linker *command_filename*

The linker processes input files in the order that it encounters them. If the linker recognizes a file as an object file, it links the file. Otherwise, it assumes that a file is a command file and begins reading and processing commands from it. Command filenames are case sensitive, regardless of the system used.

Linker Command File shows a sample linker command file called link.cmd.

Linker Command File

```
a.c.obj                    /*  First input filename          */
b.c.obj                    /*  Second input filename         */
--output_file=prog.out   /*  Option to specify output file */
--map_file=prog.map      /*  Option to specify map file    */
```

The sample file in Linker Command File contains only filenames and options. (You can place comments in a command file by delimiting them with /* and */.) To invoke the linker with this command file, enter:

```
cl7x --run_linker link.cmd
```

You can place other parameters on the command line when you use a command file:

```
cl7x --run_linker --relocatable link.cmd x.c.obj y.c.obj
```

The linker processes the command file as soon as it encounters the filename, so a.c.obj and b.c.obj are linked into the output module before x.c.obj and y.c.obj.

You can specify multiple command files. If, for example, you have a file called names.lst that contains filenames and another file called dir.cmd that contains linker directives, you could enter:

```
cl7x --run_linker names.lst dir.cmd
```

One command file can call another command file; this type of nesting is limited to 16 levels. If a command file calls another command file as input, this statement must be the *last* statement in the calling command file.

Blanks and blank lines are insignificant in a command file except as delimiters. This also applies to the format of linker directives in a command file. Command File With Linker Directives shows a sample command file that contains linker directives.

Copyright © 2026 Texas Instruments Incorporated

Command File With Linker Directives

```
a.obj b.obj c.obj           /* Input filenames       */
--output_file=prog.out      /* Options               */
--map_file=prog.map
MEMORY                      /* MEMORY directive      */
{
  FAST_MEM:  origin = 0x0100    length = 0x0100
  SLOW_MEM:  origin = 0x7000    length = 0x1000
}
SECTIONS                    /* SECTIONS directive    */
{
  .text:  > SLOW_MEM
  .data:  > SLOW_MEM
  .bss:   > FAST_MEM
}
```

For more information, see Section 12.5.4 for the MEMORY directive, and Section 12.5.5 for the SECTIONS directive.

12.5.1 Reserved Names in Linker Command Files

The following names (in both uppercase and lowercase) are reserved as keywords for linker directives. Do not use them as symbol or section names in a command file.

ADDRESS_MASK	f	LENGTH	ORG	SIZE
ALGORITHM	FILL	LOAD	ORIGIN	START
ALIGN	GROUP	LOAD_END	PAGE	TABLE
ATTR	HAMMING_MASK	LOAD_SIZE	PALIGN	TYPE
BLOCK	HIGH	LOAD_START	PARITY_MASK	UNION
COMPRESSION	INPUT_PAGE	MEMORY	RUN	UNORDERED
COPY	INPUT_RANGE	MIRRORING	RUN_END	VFILL
DSECT	l (lowercase L)	NOINIT	RUN_SIZE	
ECC	LAST	NOLOAD	RUN_START	
END	LEN	o	SECTIONS	

In addition, any section names used by the TI tools are reserved from being used as the prefix for other names, unless the section will be a subsection of the section name used by the TI tools. For example, section names may not begin with .debug.

12.5.2 Constants in Linker Command Files

You can specify constants with either of two syntax schemes: the scheme used for specifying decimal, octal, or hexadecimal constants (but not binary constants) used in the assembler or the scheme used for integer constants in C syntax. For example:

Format	Decimal	Octal	Hexadecimal
Assembler format	32	40q	020h
C format	32	040	0x20

12.5.3 Accessing Files and Libraries from a Linker Command File

Many applications use custom linker command files (or LCFs) to control the placement of code and data in target memory. For example, you may want to place a specific data object from a specific file into a specific location in target memory. This is simple to do using the available LCF syntax to reference the desired object file or library. However, a problem that many developers run into when they try to do this is a linker generated "file not found" error when accessing an object file or library from inside the LCF that has been specified earlier in the command-line invocation of the linker. Most often, this error occurs because the syntax used to access the file on the linker command-line does not match the syntax that is used to access the same file in the LCF.

Copyright © 2026 Texas Instruments Incorporated

Consider a simple example. Imagine that you have an application that requires a table of constants called "app_coeffs" to be defined in a memory area called "DDR". Assume also that the "app_coeffs" data object is defined in a .data section that resides in an object file, app_coeffs.c.obj. The app_coeffs.c.obj file is then included in the object file library app_data.lib. In your LCF, you can control the placement of the "app_coeffs" data object as follows:

```
SECTIONS
{
   ...
   .coeffs: { app_data.lib<app_coeffs.c.obj>(.data) } > DDR
   ...
}
```

Now assume that the app_data.lib object library resides in a sub-directory called "lib" relative to where you are building the application. In order to gain access to app_data.lib from the build command-line, you can use a combination of the -i and -l options to set up a directory search path which the linker can use to find the app_data.lib library:

```
%> cl7x <compile options/files> -z -i ./lib -l app_data.lib mylnk.cmd <link options/files>
```

The -i option adds the lib sub-directory to the directory search path and the -l option instructs the linker to look through the directories in the directory search path to find the app_data.lib library. However, if you do not update the reference to app_data.lib in mylnk.cmd, the linker will fail to find the app_data.lib library and generate a "file not found" error. The reason is that when the linker encounters the reference to app_data.lib inside the SECTIONS directive, there is no -l option preceding the reference. Therefore, the linker tries to open app_data.lib in the current working directory.

In essence, the linker has a few different ways of opening files:

- If there is a path specified, the linker will look for the file in the specified location. For an absolute path, the linker will try to open the file in the specified directory. For a relative path, the linker will follow the specified path starting from the current working directory and try to open the file at that location.
- If there is no path specified, the linker will try to open the file in the current working directory.
- If a -l option precedes the file reference, then the linker will try to find and open the referenced file in one of the directories in the directory search path. The directory search path is set up via -i options and environment variables (like C_DIR and C7X_C_DIR).

As long as a file is referenced in a consistent manner on the command line and throughout any applicable LCFs, the linker will be able to find and open your object files and libraries.

Returning to the earlier example, you can insert a -l option in front of the reference to app_data.lib in mylnk.cmd to ensure that the linker will find and open the app_data.lib library when the application is built:

```
SECTIONS
{
   ...
   .coeffs: { -l app_data.lib<app_coeffs.c.obj>(.data) } > DDR
   ...
}
```

Another benefit to using the -l option when referencing a file from within an LCF is that if the location of the referenced file changes, you can modify the directory search path to incorporate the new location of the file (using -i option on the command line, for example) without having to modify the LCF.

Copyright © 2026 Texas Instruments Incorporated

12.5.4 The MEMORY Directive

The linker determines where output sections are allocated into memory; it must have a model of target memory to accomplish this. The MEMORY directive allows you to specify a model of target memory so that you can define the types of memory your system contains and the address ranges they occupy. The linker maintains the model as it allocates output sections and uses it to determine which memory locations can be used for object code.

The memory configurations of C7000 systems differ from application to application. The MEMORY directive allows you to specify a variety of configurations. After you use MEMORY to define a memory model, you can use the SECTIONS directive to allocate output sections into defined memory. For more information, see Section 8.4.

The C7000 compiler requires that all code and data lie within 2 GB of virtual address space. Use a linker command file to place your code and data within this 2 GB virtual address region. The C7000 compiler produces code that uses position-independent addressing to obtain the addresses of functions (in some cases) and to access statically allocated data in memory. Instructions that use position-independent addressing have limited reach.

12.5.4.1 Default Memory Model

If you do not use the MEMORY directive, the linker uses a default memory model that may be suitable for running executables on a generic simulator. Please consult your device documentation to determine what memory ranges are available. This model assumes that the full 48-bit address space (2^{48} locations) is present in the system and available for use. For more information about the default memory model, see Section 12.7.

12.5.4.2 MEMORY Directive Syntax

The MEMORY directive identifies ranges of memory that are physically present in the target system and can be used by a program. Each range has several characteristics:

- Name
- Starting address
- Length
- Optional set of attributes
- Optional fill specification

When you use the MEMORY directive, be sure to identify all memory ranges that are available for the program to access at run time. Memory defined by the MEMORY directive is configured; any memory that you do not explicitly account for with MEMORY is unconfigured. The linker does not place any part of a program into unconfigured memory. You can represent nonexistent memory spaces by simply not including an address range in a MEMORY directive statement.

The MEMORY directive is specified in a command file by the word MEMORY (uppercase), followed by a list of memory range specifications enclosed in braces. The MEMORY directive in the example that follows defines a system that has 4K bytes of fast external memory at address 0x0000 0000, 2K bytes of slow external memory at address 0x0000 1000 and 4K bytes of slow external memory at address 0x1000 0000. It also demonstrates the use of memory range expressions as well as start/end/size address operators (see Section 12.5.4.3).

The MEMORY Directive

```
/*********************************************************/
/*      Sample command file with MEMORY directive        */
/*********************************************************/
file1.c.obj file2.c.obj              /*    Input files     */
--output_file=prog.out            /*    Options         */
#define BUFFER 0
MEMORY
{
   FAST_MEM (RX): origin = 0x00000000    length = 0x00001000 + BUFFER
   SLOW_MEM (RW): origin = end(FAST_MEM) length = 0x00001800 - size(FAST_MEM)
   EXT_MEM  (RX): origin = 0x10000000    length = size(FAST_MEM)
```


Copyright © 2026 Texas Instruments Incorporated

The general syntax for the MEMORY directive is:

MEMORY

{

name 1 [**(***attr* **)**] **: origin =** *expr* **, length =** *expr* [**, fill =** *constant*] [**LAST(***sym* **)**]

.

.

name n [**(***attr* **)**] **: origin =** *expr* **, length =** *expr* [**, fill =** *constant*] [**LAST(***sym* **)**]

}

name	names a memory range. A memory name can be one to 64 characters; valid characters include A-Z, a-z, $, ., and _. The names have no special significance to the linker; they simply identify memory ranges. Memory range names are internal to the linker and are not retained in the output file or in the symbol table. All memory ranges must have unique names and must not overlap.
attr	specifies one to four attributes associated with the named range. Attributes are optional; when used, they must be enclosed in parentheses. Attributes restrict the allocation of output sections into certain memory ranges. If you do not use any attributes, you can allocate any output section into any range with no restrictions. Any memory for which no attributes are specified (including all memory in the default model) has all four attributes. Valid attributes are: **R** specifies that the memory can be read. **W** specifies that the memory can be written to. **X** specifies that the memory can contain executable code. **I** specifies that the memory can be initialized.
origin	specifies the starting address of a memory range; enter as *origin*, *org*, or *o*. The value, specified in bytes, is a 64-bit integer constant expression, which can be decimal, octal, or hexadecimal.
length	specifies the length of a memory range; enter as *length*, *len*, or *l*. The value, specified in bytes, is a 64-bit integer constant expression, which can be decimal, octal, or hexadecimal.
fill	specifies a fill character for the memory range; enter as *fill* or *f*. Fills are optional. The value is an integer constant and can be decimal, octal, or hexadecimal. The fill value is used to fill areas of the memory range that are not allocated to a section.
LAST	optionally specifies a symbol that can be used at run-time to find the address of the last allocated byte in the memory range. See Section 12.5.9.8.

Note

Filling Memory Ranges: If you specify fill values for large memory ranges, your output file will be very large because filling a memory range (even with 0s) causes raw data to be generated for all unallocated blocks of memory in the range.

The following example specifies a memory range with the R and W attributes and a fill constant of 0FFFFFFFFh:

```
MEMORY
{
RFILE (RW) : o = 0x00000020, l = 0x00001000, f = 0xFFFFFFFF
}
```

You normally use the MEMORY directive in conjunction with the SECTIONS directive to control placement of output sections. For more information about the SECTIONS directive, see Section 12.5.5.

12.5.4.3 Expressions and Address Operators

Memory range origin and length can use expressions of integer constants with the following operators:

Binary operators:	* / % + - << >> == = < <= > >= & | && ||
Unary operators:	- ~ !

Expressions are evaluated using standard C operator precedence rules.

No checking is done for overflow or underflow, however, expressions are evaluated using a larger integer type.

Copyright © 2026 Texas Instruments Incorporated

Preprocess directive #define constants can be used in place of integer constants. Global symbols cannot be used in Memory Directive expressions.

Three address operators reference memory range properties from prior memory range entries:

START(MR)	Returns start address for previously defined memory range MR.
SIZE(MR)	Returns size of previously defined memory range MR.
END(MR)	Returns end address for previously defined memory range MR.

Origin and Length as Expressions

```
/*********************************************************/
/*      Sample command file with MEMORY directive        */
/*********************************************************/
file1.c.obj file2.c.obj              /*    Input files     */
--output_file=prog.out            /*    Options         */
#define ORIGIN 0x00000000
#define BUFFER 0x00000200
#define CACHE  0x0001000
MEMORY
{
   FAST_MEM (RX): origin = ORIGIN + CACHE length = 0x00001000 + BUFFER
   SLOW_MEM (RW): origin = end(FAST_MEM)  length = 0x00001800 - size(FAST_MEM)
   EXT_MEM  (RX): origin = 0x10000000     length = size(FAST_MEM) - CACHE
```

12.5.5 The SECTIONS Directive

After you use MEMORY to specify the target system's memory model, you can use SECTIONS to place output sections into specific named memory ranges or into memory that has specific attributes. For example, you could allocate the .text and .data sections into the area named FAST_MEM and allocate the .bss section into the area named SLOW_MEM.

The SECTIONS directive controls your sections in the following ways:

- Describes how input sections are combined into output sections
- Defines output sections in the executable program
- Allows you to control where output sections are placed in memory in relation to each other and to the entire memory space (Note that the memory placement order is *not* simply the sequence in which sections occur in the SECTIONS directive.)
- Permits renaming of output sections

For more information, see Section 8.4. Subsections allow you to manipulate sections with greater precision.

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections. Section 12.7 describes this algorithm in detail.

12.5.5.1 SECTIONS Directive Syntax

The SECTIONS directive is specified in a command file by the word SECTIONS (uppercase), followed by a list of output section specifications enclosed in braces.

The general syntax for the SECTIONS directive is:

SECTIONS

{

name : [*property* [, **property**] [, **property**] . . .]

name : [*property* [, **property**] [, **property**] . . .]

name : [*property* [, **property**] [, **property**] . . .]

}

Copyright © 2026 Texas Instruments Incorporated

Each section specification, beginning with *name*, defines an output section. (An output section is a section in the output file.) Section names can refer to sections, subsections, or archive library members. (See Section 12.5.5.4 for information on multi-level subsections.) After the section name is a list of properties that define the section's contents and how the section is allocated. The properties can be separated by optional commas. Possible properties for a section are as follows:

- **Load allocation** defines where in memory the section is to be loaded. See Section 9.5 and Section 12.5.6.

  Syntax: **load =** *allocation* or
  > *allocation*

- **Run allocation** defines where in memory the section is to be run.

  Syntax: **run =** *allocation* or
  run > *allocation*

- **Input sections** defines the input sections (object files) that constitute the output section. See Section 12.5.5.3.

  Syntax: **{** *input_sections* **}**

- **Section type** defines flags for special section types. See Section 12.5.8.

  Syntax: **type = COPY** or
  type = DSECT or

- **Fill value** defines the value used to fill uninitialized holes. See Section 12.5.10.

  Syntax: **fill =** *value*

The following example shows a SECTIONS directive in a sample linker command file.

```
/************************************************/
/*  Sample command file with SECTIONS directive   */
/************************************************/
file1.c.obj    file2.c.obj           /* Input files   */
--output_file=prog.out           /* Options       */
SECTIONS
{
   .text:       load = EXT_MEM, run = 0x00000800
   .const:      load = FAST_MEM
   .bss:        load = SLOW_MEM
   .vectors:    load = 0x00000000
      {
         t1.c.obj(.intvec1)
         t2.c.obj(.intvec2)
         endvec = .;
      }
   .data:alpha: align = 16
   .data:beta:  align = 16
}
```

Figure 12-2 shows the output sections defined by the SECTIONS directive in the previous example (.vectors, .text, .const, .bss, .data:alpha, and .data:beta) and shows how these sections are allocated in memory using the MEMORY directive given in Section 12.5.4.2.

Copyright © 2026 Texas Instruments Incorporated

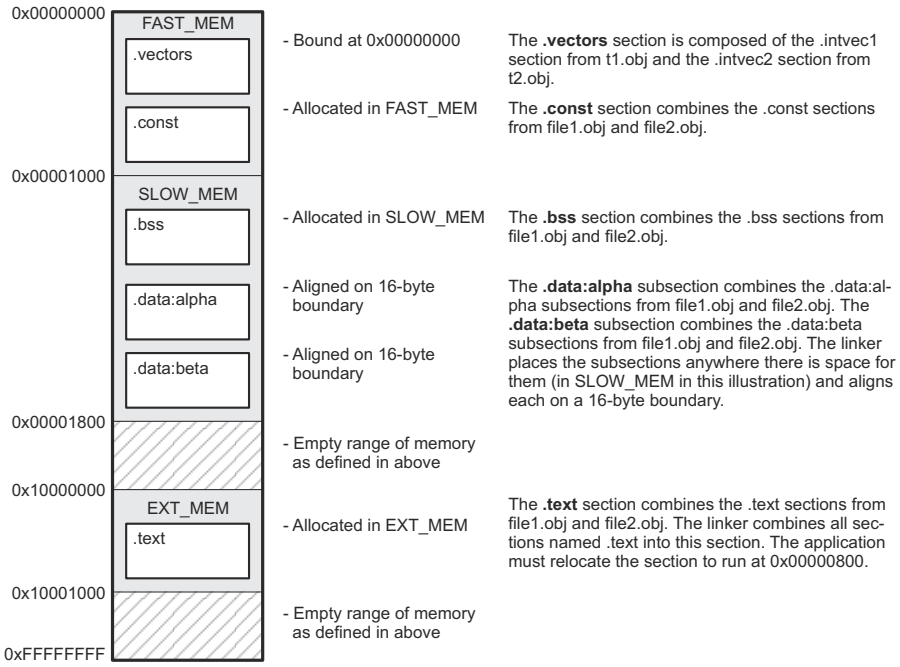


Figure 12-2. Section Placement Defined by the SECTIONS Directive Example

12.5.5.2 Section Allocation and Placement

The linker assigns each output section two locations in target memory: the location where the section will be loaded and the location where it will be run. Usually, these are the same, and you can think of each section as having only a single address. The process of locating the output section in the target's memory and assigning its address(es) is called placement. For more information about using separate load and run placement, see Section 12.5.6.

If you do not tell the linker how to allocate a section, it uses a default algorithm to place the section. Generally, the linker puts sections wherever they fit into configured memory. You can override the default placement for a section by defining it within a SECTIONS directive and providing instructions on how to allocate it.

You control placement by specifying one or more allocation parameters. Each parameter consists of a keyword, an optional equal sign or greater-than sign, and a value optionally enclosed in parentheses. If load and run placement are separate, all parameters following the keyword LOAD apply to load placement, and those following the keyword RUN apply to run placement. The allocation parameters are:

Binding allocates a section at a specific address.

```
.text: load = 0x1000
```

Named memory allocates the section into a range defined in the MEMORY directive with the specified name (like SLOW_MEM) or attributes.

```
.text: load > SLOW_MEM
```

Alignment uses the align or palign keyword to specify the section must start on an address boundary.

```
.text: align = 0x100
```

Blocking uses the block keyword to specify the section must fit between two address aligned to the blocking factor. If a section is too large, it starts on an address boundary.

```
.text: block(0x100)
```


Copyright © 2026 Texas Instruments Incorporated

For the load (usually the only) allocation, use a greater-than sign and omit the load keyword:

```
    .text: > SLOW_MEM
    .text: {...} > SLOW_MEM
    .text: > 0x4000
```

If more than one parameter is used, you can string them together as follows:

```
    .text: > SLOW_MEM align 16
```

Or if you prefer, use parentheses for readability:

```
    .text: load = (SLOW_MEM align(16))
```

You can also use an input section specification to identify the sections from input files that are combined to form an output section. See Section 12.5.5.3.

12.5.5.2.1 Binding

You can set the starting address for an output section by following the section name with an address:

```
    .text: 0x00001000
```

This example specifies that the .text section must begin at location 0x1000. The binding address must be a 64-bit constant.

Output sections can be bound anywhere in configured memory (assuming there is enough space), but they cannot overlap. If there is not enough space to bind a section to a specified address, the linker issues an error message.

Note

Binding is Incompatible With Alignment and Named Memory: You cannot bind a section to an address if you use alignment or named memory. If you try this, the linker issues an error message.

12.5.5.2.2 Named Memory

You can allocate a section into a memory range that is defined by the MEMORY directive (see Section 12.5.4). This example names ranges and links sections into them:

```
MEMORY
{
   SLOW_MEM (RIX)  : origin = 0x00000000,  length = 0x00001000
   FAST_MEM (RWIX) : origin = 0x03000000,  length = 0x00000300
}
SECTIONS
{
   .text  :   > SLOW_MEM
   .data  :   > FAST_MEM ALIGN(128)
   .bss   :   > FAST_MEM
}
```

In this example, the linker places .text into the area called SLOW_MEM. The .data and .bss output sections are allocated into FAST_MEM. You can align a section within a named memory range; the .data section is aligned on a 128-byte boundary within the FAST_MEM range.

Copyright © 2026 Texas Instruments Incorporated

Similarly, you can link a section into an area of memory that has particular attributes. To do this, specify a set of attributes (enclosed in parentheses) instead of a memory name. Using the same MEMORY directive declaration, you can specify:

```
SECTIONS
{
   .text: > (X)   /* .text --> executable memory      */
   .data: > (RI)  /* .data --> read or init memory    */
   .bss : > (RW)  /* .bss --> read or write memory    */
}
```

In this example, the .text output section can be linked into either the SLOW_MEM or FAST_MEM area because both areas have the X attribute. The .data section can also go into either SLOW_MEM or FAST_MEM because both areas have the R and I attributes. The .bss output section, however, must go into the FAST_MEM area because only FAST_MEM is declared with the W attribute.

You cannot control where in a named memory range a section is allocated, although the linker uses lower memory addresses first and avoids fragmentation when possible. In the preceding examples, assuming no conflicting assignments exist, the .text section starts at address 0. If a section must start on a specific address, use binding instead of named memory.

12.5.5.2.3 Controlling Placement Using The HIGH Location Specifier

The linker allocates output sections from low to high addresses within a designated memory range by default. Alternatively, you can cause the linker to allocate a section from high to low addresses within a memory range by using the HIGH location specifier in the SECTION directive declaration. You might use the HIGH location specifier in order to keep RTS code separate from application code, so that small changes in the application do not cause large changes to the memory map.

For example, given this MEMORY directive:

```
MEMORY
{
    RAM             : origin = 0x0200, length = 0x0800
    FLASH           : origin = 0x1100, length = 0xEEE0
    VECTORS         : origin = 0xFFE0, length = 0x001E
    RESET           : origin = 0xFFFE, length = 0x0002
}
```

and an accompanying SECTIONS directive:

```
SECTIONS
{
    .bss     : {} > RAM
    .sysmem  : {} > RAM
    .stack   : {} > RAM (HIGH)
}
```


Copyright © 2026 Texas Instruments Incorporated

The HIGH specifier used on the .stack section placement causes the linker to attempt to allocate .stack into the higher addresses within the RAM memory range. The .bss and sections are allocated into the lower addresses within RAM. Example 12-1 illustrates a portion of a map file that shows where the given sections are allocated within RAM for a typical program.

Example 12-1. Linker Placement With the HIGH Specifier

```
.bss        0    00000200    00000270     UNINITIALIZED
                   00000200    0000011a       rtsxxx.lib  : defs.c.obj (.bss)
                   0000031a    00000088                   : trgdrv.c.obj (.bss)
                   000003a2    00000078                   : lowlev.c.obj (.bss)
                   0000041a    00000046                   : exit.c.obj (.bss)
                   00000460    00000008                   : memory.c.obj (.bss)
                   00000468    00000004                   : _lock.c.obj (.bss)
                   0000046c    00000002                   : fopen.c.obj (.bss)
                   0000046e    00000002       hello.c.obj (.bss)
.sysmem     0    00000470    00000120     UNINITIALIZED
                   00000470    00000004       rtsxxx .lib : memory.c.obj (.sysmem)
.stack      0    000008c0    00000140     UNINITIALIZED
                   000008c0    00000002       rtsxxx .lib : boot.c.obj (.stack)
```

As shown in Example 12-1 , the .bss and .sysmem sections are allocated at the lower addresses of RAM (0x0200 - 0x0590) and the .stack section is allocated at address 0x08c0, even though lower addresses are available.

Without using the HIGH specifier, the linker allocation would result in the code shown in Example 12-2

The HIGH specifier is ignored if it is used with specific address binding or automatic section splitting (>> operator).

Example 12-2. Linker Placement Without HIGH Specifier

```
.bss        0    00000200    00000270     UNINITIALIZED
                   00000200    0000011a       rtsxxx.lib  : defs.c.obj (.bss)
                   0000031a    00000088                   : trgdrv.c.obj (.bss)
                   000003a2    00000078                   : lowlev.c.obj (.bss)
                   0000041a    00000046                   : exit.c.obj (.bss)
                   00000460    00000008                   : memory.c.obj (.bss)
                   00000468    00000004                   : _lock.c.obj (.bss)
                   0000046c    00000002                   : fopen.c.obj (.bss)
                   0000046e    00000002       hello.c.obj (.bss)
.stack      0    00000470    00000140     UNINITIALIZED
                   00000470    00000002       rtsxxx.lib  : boot.c.obj (.stack)
.sysmem     0    000005b0    00000120     UNINITIALIZED
                   000005b0    00000004       rtsxxx.lib  : memory.c.obj (.sysmem)
```

12.5.5.2.4 Alignment and Blocking

You can tell the linker to place an output section at an address that falls on an n-byte boundary, where n is a power of 2, by using the align keyword. For example, the following code allocates .text so that it falls on a 32-byte boundary:

```
.text: load = align(32)
```

Blocking is a weaker form of alignment that allocates a section anywhere *within* a block of size n. The specified block size must be a power of 2. For example, the following code allocates .bss so that the entire section is contained in a single 128-byte or begins on that boundary:

You can use alignment or blocking alone or in conjunction with a memory area, but alignment and blocking cannot be used together.

12.5.5.2.5 Alignment With Padding

As with align, you can tell the linker to place an output section at an address that falls on an n-byte boundary, where n is a power of 2, by using the palign keyword. In addition, palign ensures that the size of the section is a multiple of its placement alignment restrictions, padding the section size up to such a boundary, as needed.

Copyright © 2026 Texas Instruments Incorporated

For example, the following code lines allocate .text on a 2-byte boundary within the PMEM area. The .text section size is guaranteed to be a multiple of 2 bytes. Both statements are equivalent:

```
   .text: palign(2) {} > PMEM
   .text: palign = 2 {} > PMEM
```

If the linker adds padding to an initialized output section then the padding space is also initialized. By default, padding space is filled with a value of 0 (zero). However, if a fill value is specified for the output section then any padding for the section is also filled with that fill value. For example, consider the following section specification:

```
   .mytext: palign(8), fill = 0xffffffff {} > PMEM
```

In this example, the length of the .mytext section is before the palign operator is applied. The contents of .mytext are as follows:

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
```

After the palign operator is applied, the length of .mytext is 8 bytes, and its contents are as follows:

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
0006 0xffff
```

The size of .mytext has been bumped to a multiple of 8 bytes and the padding created by the linker has been filled with 0xff.

The fill value specified in the linker command file is interpreted as a 16-bit constant. If you specify this code:

```
   .mytext: palign(8), fill = 0xff {} > PMEM
```

The fill value assumed by the linker is 0x00ff, and .mytext will then have the following contents:

```
addr content
---- -------
0000 0x1234
0002 0x1234
0004 0x1234
0006 0x00ff
```

If the palign operator is applied to an uninitialized section, then the size of the section is bumped to the appropriate boundary, as needed, but any padding created is not initialized.

The palign operator can also take a parameter of *power2*. This parameter tells the linker to add padding to increase the section's size to the next power of two boundary. In addition, the section is aligned on that power of 2 as well. For example, consider the following section specification:

```
    .mytext: palign(power2) {} > PMEM
```

Assume that the size of the .mytext section is 120 bytes and PMEM starts at address 0x10020. After applying the palign(power2) operator, the .mytext output section will have the following properties:

```
  name     addr          size     align
-------    ----------    -----    -----
.mytext    0x00010080    0x80     128
```


Copyright © 2026 Texas Instruments Incorporated

12.5.5.3 Specifying Input Sections

An input section specification identifies the sections from input files that are combined to form an output section. In general, the linker combines input sections by concatenating them in the order in which they are specified. However, if alignment or blocking is specified for an input section, all of the input sections within the output section are ordered as follows:

- All aligned sections, from largest to smallest
- All blocked sections, from largest to smallest
- All other sections, from largest to smallest

The size of an output section is the sum of the sizes of the input sections that it comprises.

Example 12-3 shows the most common type of section specification; note that no input sections are listed.

Example 12-3. The Most Common Method of Specifying Section Contents

```
SECTIONS
{
   .text:
   .data:
   .bss:
}
```

In Example 12-3, the linker combines all .text sections from the input files into the .text output section. The linker concatenates the .text input sections in the order it encounters them in the input files. The linker performs similar operations with the .data and .bss sections. You can use this type of specification for any output section.

You can explicitly specify the input sections to form an output section. Each input section is identified by its filename and section name. If a filename is hyphenated (or contains special characters), enclose it in quotes:

```
SECTIONS
{
  .text :              /* Build .text output section        */
  {
    f1.c.obj(.text)      /* Link .text section from f1.c.obj    */
    f2.c.obj(sec1)       /* Link sec1 section from f2.c.obj     */
    "f3-new.c.obj"       /* Link ALL sections from f3-new.c.obj */
    f4.c.obj"(.text,sec2) /* Link .text and sec2 from f4.c.obj   */
   }
}
```

It is not necessary for input sections to have the same name as each other or as the output section they become part of. If a file is listed with no sections, *all* of its sections are included in the output section. If any additional input sections have the same name as an output section but are not explicitly specified by the SECTIONS directive, they are automatically linked in at the end of the output section. For example, if the linker found more .text sections in the preceding example and these .text sections *were not* specified anywhere in the SECTIONS directive, the linker would concatenate these extra sections after f4.c.obj(sec2).

Copyright © 2026 Texas Instruments Incorporated

The specifications in Example 12-3 are actually a shorthand method for the following:

```
SECTIONS
{
  .text: { *(.text) }
  .data: { *(.data) }
  .bss:  { *(.bss)  }
}
```

The specification *(.text) means *the unallocated .text sections from all input files.* This format is useful if:

- You want the output section to contain all input sections that have a specified name, but the output section name is different from the input sections' name.
- You want the linker to allocate the input sections before it processes additional input sections or commands within the braces.

The following example illustrates the two purposes above:

```
SECTIONS
{
   .text  :  {
                  abc.c.obj(xqt)
                  *(.text)            }
   .data  :  {
                  *(.data)
                  fil.c.obj(table)    }
}
```

In this example, the .text output section contains a named section xqt from file abc.c.obj, which is followed by all the .text input sections. The .data section contains all the .data input sections, followed by a named section table from the file fil.c.obj. This method includes all the unallocated sections. For example, if one of the .text input sections was already included in another output section when the linker encountered *(.text), the linker could not include that first .text input section in the second output section.

Each input section acts as a prefix to gather longer-named sections. For example, the pattern *(.data) matches .dataspecial. Thus, prefixes enable subsections, which are described in the following section.

12.5.5.4 Using Multi-Level Subsections

Subsections can be identified with a base section name and one or more subsection names separated by colons. For example, A:B and A:B:C name subsections of base section A. In certain places, specifying a base name, such as A, selects section A as well as any subsections of A, such as A:B or A:C:D.

A name such as A:B can specify a (sub)section of that name as well as any (multi-level) subsections beginning with that name, such as A:B:C, A:B:OTHER, etc. All subsections of A:B are also subsections of A. A and A:B are supersections of A:B:C. Among a group of supersections of a subsection, the nearest supersection is the supersection with the longest name. Thus, among {A, A:B} the nearest supersection of A:B:C:D is A:B. With multiple levels of subsections, the constraints are the following:

1. When specifying **input** sections within a file (or library unit) the section name selects an input section of the same name and any subsections of that name.
2. Input sections that are not explicitly allocated are allocated in an existing **output** section of the same name or in the nearest existing supersection of such an output section. An exception to this rule is that during a partial link (specified by the --relocatable linker option) a subsection is allocated only to an existing output section of the same name.
3. If no such output section described in 2) is defined, the input section is put in a **newly created output** section with the same name as the base name of the input section

Copyright © 2026 Texas Instruments Incorporated

Consider linking input sections with the following names:

europe:north:norway	europe:central:france	europe:south:spain
europe:north:sweden	europe:central:germany	europe:south:italy
europe:north:finland	europe:central:denmark	europe:south:malta
europe:north:iceland		

This SECTIONS specification allocates the input sections as indicated in the comments:

```
SECTIONS {
  nordic:  {*(europe:north)
            *(europe:central:denmark)} /* the nordic countries */
  central: {*(europe:central)}         /* france, germany      */
  therest: {*(europe)}                 /* spain, italy, malta  */
}
```

This SECTIONS specification allocates the input sections as indicated in the comments:

```
SECTIONS {
  islands: {*(europe:south:malta)
            *(europe:north:iceland)}  /* malta, iceland   */
  europe:north:finland : {}           /* finland          */
  europe:north         : {}           /* norway, sweden   */
  europe:central       : {}           /* germany, denmark */
  europe:central:france: {}           /* france           */
  /* (italy, spain) go into a linker-generated output section "europe" */
}
```

Note

Upward Compatibility of Multi-Level Subsections: Existing linker commands that use the existing single-level subsection features and which do not contain section names containing multiple colon characters continue to behave as before. However, if section names in a linker command file or in the input sections supplied to the linker contain multiple colon characters, some change in behavior could be possible. You should carefully consider the impact of the rules for multiple levels to see if it affects a particular system link.

12.5.5.5 Specifying Library or Archive Members as Input to Output Sections

You can specify one or more members of an object library or archive for input to an output section. Consider this SECTIONS directive:

Example 12-4. Archive Members to Output Sections

```
SECTIONS
{
boot>BOOT1
{
-l rtsXX.lib<boot.c.obj> (.text)
-l rtsXX.lib<exit.c.obj strcpy.c.obj> (.text)
}
.rts>BOOT2
{
-l rtsXX.lib (.text)
}
.text>RAM
{
* (.text)
}
}
```

In Example 12-4, the .text sections of boot.c.obj, exit.c.obj, and strcpy.c.obj are extracted from the run-time-support library and placed in the .boot output section. The remainder of the run-time-support library object that is

Copyright © 2026 Texas Instruments Incorporated

referenced is allocated to the .rts output section. Finally, the remainder of all other .text sections are to be placed in section .text.

An archive member or a list of members is specified by surrounding the member name(s) with angle brackets < and > after the library name. Any object files separated by commas or spaces from the specified archive file are legal within the angle brackets.

The --library option (which normally implies a library path search be made for the named file following the option) listed before each library in Example 12-4 is optional when listing specific archive members inside < >. Using < > implies that you are referring to a library.

To collect a set of the input sections from a library in one place, use the --library option within the SECTIONS directive. For example, the following collects all the .text sections from rts7100_le.lib into the .rtstest section:

```
SECTIONS
{
   .rtstest { -l rts7100_le.lib(.text) } > RAM
}
```

Note

SECTIONS Directive Effect on --priority: Specifying a library in a SECTIONS directive causes that library to be entered in the list of libraries that the linker searches to resolve references. If you use the --priority option, the first library specified in the command file will be searched first.

12.5.5.6 Allocation Using Multiple Memory Ranges

The linker allows you to specify an explicit list of memory ranges into which an output section can be allocated. Consider the following example:

```
MEMORY
{
   P_MEM1 :  origin = 0x02000,  length = 0x01000
   P_MEM2 :  origin = 0x04000,  length = 0x01000
   P_MEM3 :  origin = 0x06000,  length = 0x01000
   P_MEM4 :  origin = 0x08000,  length = 0x01000
}
SECTIONS
{
   .text  : { } > P_MEM1 | P_MEM2 | P_MEM4
}
```

The | operator is used to specify the multiple memory ranges. The .text output section is allocated as a whole into the first memory range in which it fits. The memory ranges are accessed in the order specified. In this example, the linker first tries to allocate the section in P_MEM1. If that attempt fails, the linker tries to place the section into P_MEM2, and so on. If the output section is not successfully allocated in any of the named memory ranges, the linker issues an error message.

With this type of SECTIONS directive specification, the linker can seamlessly handle an output section that grows beyond the available space of the memory range in which it is originally allocated. Instead of modifying the linker command file, you can let the linker move the section into one of the other areas.

Copyright © 2026 Texas Instruments Incorporated

12.5.5.7 Automatic Splitting of Output Sections Among Non-Contiguous Memory Ranges

The linker can split output sections among multiple memory ranges for efficient allocation. Use the >> operator to indicate that an output section can be split, if necessary, into the specified memory ranges:

```
MEMORY
{
   P_MEM1 :  origin = 0x2000,  length = 0x1000
   P_MEM2 :  origin = 0x4000,  length = 0x1000
   P_MEM3 :  origin = 0x6000,  length = 0x1000
   P_MEM4 :  origin = 0x8000,  length = 0x1000
}
SECTIONS
{
  .text: { *(.text) } >> P_MEM1 | P_MEM2 | P_MEM3 | P_MEM4
}
```

In this example, the >> operator indicates that the .text output section can be split among any of the listed memory areas. If the .text section grows beyond the available memory in P_MEM1, it is split on an input section boundary, and the remainder of the output section is allocated to P_MEM2 | P_MEM3 | P_MEM4.

The | operator is used to specify the list of multiple memory ranges.

You can also use the >> operator to indicate that an output section can be split within a single memory range. This functionality is useful when several output sections must be allocated into the same memory range, but the restrictions of one output section cause the memory range to be partitioned. Consider the following example:

```
MEMORY
{
   RAM :  origin = 0x1000,  length = 0x8000
}
SECTIONS
{
  .special: { f1.c.obj(.text) } load = 0x4000
  .text: { *(.text) } >> RAM
}
```

The .special output section is allocated near the middle of the RAM memory range. This leaves two unused areas in RAM: from 0x1000 to 0x4000, and from the end of f1.c.obj(.text) to 0x8000. The specification for the .text section allows the linker to split the .text section around the .special section and use the available space in RAM on either side of .special.

The >> operator can also be used to split an output section among all memory ranges that match a specified attribute combination. For example:

```
MEMORY
{
  P_MEM1 (RWX) : origin = 0x1000,  length = 0x2000
  P_MEM2 (RWI) : origin = 0x4000,  length = 0x1000
}
SECTIONS
{
  .text: { *(.text) } >> (RW)
}
```

The linker attempts to allocate all or part of the output section into any memory range whose attributes match the attributes specified in the SECTIONS directive.

This SECTIONS directive has the same effect as:

```
SECTIONS
{
.text: { *(.text) } >> P_MEM1 | P_MEM2}
}
```

Certain sections should not be split:

Copyright © 2026 Texas Instruments Incorporated

- Certain sections created by the compiler, including
  - The .cinit section, which contains the autoinitialization table for C/C++ programs
  - The .pinit section, which contains the list of global constructors for C++ programs
  - The .bss section, which defines global variables
- An output section with an input section specification that includes an expression to be evaluated. The expression may define a symbol that is used in the program to manage the output section at run time.
- An output section that has a START(), END(), OR SIZE() operator applied to it. These operators provide information about a section's load or run address, and size. Splitting the section may compromise the integrity of the operation.
- The run allocation of a UNION. (Splitting the load allocation of a UNION is allowed.)

If you use the >> operator on any of these sections, the linker issues a warning and ignores the operator.

12.5.6 Placing a Section at Different Load and Run Addresses

At times, you may want to load code into one area of memory and run it in another. For example, you may have performance-critical code in slow external memory. The code must be loaded into slow external memory, but it would run faster in fast external memory.

The linker provides a simple way to accomplish this. You can use the SECTIONS directive to direct the linker to allocate a section twice: once to set its load address and again to set its run address. For example:

```
    .fir: load = SLOW_MEM, run = FAST_MEM
```

Use the *load* keyword for the load address and the *run* keyword for the run address.

See Section 9.5 for an overview on run-time relocation.

The application must copy the section from its load address to its run address; this does *not* happen automatically when you specify a separate run address. (The TABLE operator instructs the linker to produce a copy table; see Section 12.8.4.1.)

12.5.6.1 Specifying Load and Run Addresses

The load address determines where a loader places the raw data for the section. Any references to the section (such as labels in it) refer to its run address.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and loads and runs at the same address. If you provide both allocations, the section is allocated as if it were two sections of the same size. This means that both allocations occupy space in the memory map and cannot overlay each other or other sections. (The UNION directive provides a way to overlay sections; see Section 12.5.7.2.)

If either the load or run address has additional parameters, such as alignment or blocking, list them after the appropriate keyword. Everything related to allocation after the keyword *load* affects the load address until the keyword *run* is seen, after which, everything affects the run address. The load and run allocations are completely independent, so any qualification of one (such as alignment) has no effect on the other. You can also specify run first, then load. Use parentheses to improve readability.

The examples that follow specify load and run addresses.

In this example, align applies only to load:

```
    .data: load = SLOW_MEM, align = 32, run = FAST_MEM
```

The following example uses parentheses, but has effects that are identical to the previous example:

```
    .data: load = (SLOW_MEM align 32), run = FAST_MEM
```


Copyright © 2026 Texas Instruments Incorporated

The following example aligns FAST_MEM to 32 bits for run allocations and aligns all load allocations to 16 bits:

```
    .data: run  = FAST_MEM, align 32, load = align 16
```

For more information on run-time relocation see Section 9.5.

Uninitialized sections (such as .bss) are not loaded, so their only significant address is the run address. The linker allocates uninitialized sections only once: if you specify both run and load addresses, the linker warns you and ignores the load address. Otherwise, if you specify only one address, the linker treats it as a run address, regardless of whether you call it load or run.

This example specifies load and run addresses for an uninitialized section:

```
    .bss: load = 0x1000, run = FAST_MEM
```

A warning is issued, load is ignored, and space is allocated in FAST_MEM. All of the following examples have the same effect. The .bss section is allocated in FAST_MEM.

```
    .dbss: load = FAST_MEM
    .bss: run = FAST_MEM
    .bss: > FAST_MEM
```

12.5.7 Using GROUP and UNION Statements

Two SECTIONS statements allow you to organize or conserve memory: GROUP and UNION. Grouping sections causes the linker to allocate them contiguously in memory. Unioning sections causes the linker to allocate them to the same run address.

12.5.7.1 Grouping Output Sections Together

The SECTIONS directive's GROUP option forces several output sections to be allocated contiguously and in the order listed, unless the UNORDERED operator is used. For example, assume that a section named term_rec contains a termination record for a table in the .data section. You can force the linker to allocate .data and term_rec together:

Allocate Sections Together

```
SECTIONS
{
   .text           /* Normal output section             */
   .bss            /* Normal output section             */
   GROUP 0x00001000 : /* Specify a group of sections    */
   {
      .data        /* First section in the group        */
      term_rec     /* Allocated immediately after .data */
   }
}
```

You can use binding, alignment, or named memory to allocate a GROUP in the same manner as a single output section. In the preceding example, the GROUP is bound to address 0x1000. This means that .data is allocated at 0x1000, and term_rec follows it in memory.

Note

You Cannot Specify Addresses for Sections Within a GROUP: When you use the GROUP option, binding, alignment, or allocation into named memory can be specified for the group only. You cannot use binding, named memory, or alignment for sections within a group.

12.5.7.2 Overlaying Sections With the UNION Statement

For some applications, you may want to allocate more than one section that occupies the same address during run time. For example, you may have several routines you want in fast external memory at different stages

Copyright © 2026 Texas Instruments Incorporated

of execution. Or you may want several data objects that are not active at the same time to share a block of memory. The UNION statement within the SECTIONS directive provides a way to allocate several sections at the same run-time address.

In The UNION Statement, the .bss sections from file1.c.obj and file2.c.obj are allocated at the same address in FAST_MEM. In the memory map, the union occupies as much space as its largest component. The components of a union remain independent sections; they are simply allocated together as a unit.

The UNION Statement

```
SECTIONS
{
   .text: load = SLOW_MEM
   UNION: run  = FAST_MEM
   {
      .bss:part1: { file1.c.obj(.bss) }
      .bss:part2: { file2.c.obj(.bss) }
   }
      .bss:part3: run = FAST_MEM { globals.c.obj(.bss) }
}
```

Allocating a section as part of a union affects only its *run address.* Sections can *never* be overlaid for loading. If an initialized section is a union member (an initialized section, such as .text, has raw data), its load allocation *must* be separately specified. See Separate Load Addresses for UNION Sections. (There is an exception to this rule when combining an initialized section with uninitialized sections; see Section 12.5.7.3.)

Separate Load Addresses for UNION Sections

```
UNION run = FAST_MEM
{
   .text:part1: load = SLOW_MEM, { file1.c.obj(.text) }
   .text:part2: load = SLOW_MEM, { file2.c.obj(.text) }
}
```

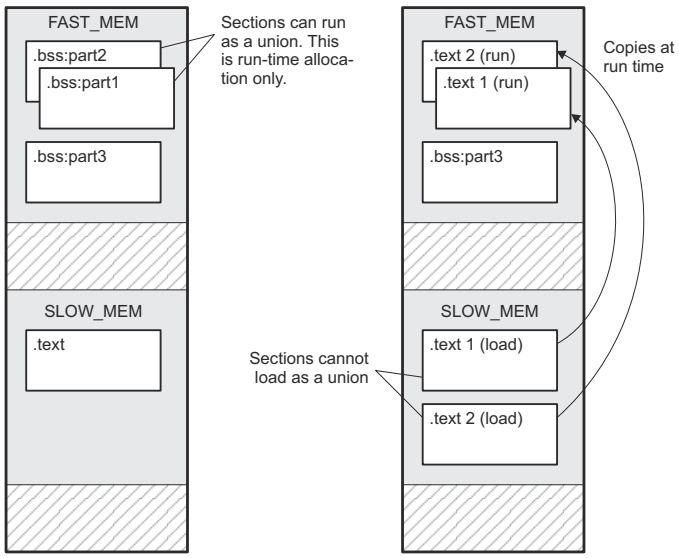


Figure 12-3. Memory Allocation Shown in The UNION Statement and Separate Load Addresses for UNION Sections

Since the .text sections contain raw data, they cannot *load* as a union, although they can be *run* as a union. Therefore, each requires its own load address. If you fail to provide a load allocation for an initialized section within a UNION, the linker issues a warning and allocates load space anywhere it can in configured memory.

Uninitialized sections are not loaded and do not require load addresses.

Copyright © 2026 Texas Instruments Incorporated

The UNION statement applies only to allocation of run addresses, so it is meaningless to specify a load address for the union itself. For purposes of allocation, the union is treated as an uninitialized section: any one allocation specified is considered a run address, and if both run and load addresses are specified, the linker issues a warning and ignores the load address.

12.5.7.3 Using Memory for Multiple Purposes

One way to reduce an application's memory requirement is to use the same range of memory for multiple purposes. You can first use a range of memory for system initialization and startup. Once that phase is complete, the same memory can be repurposed as a collection of uninitialized data variables or a heap. To implement this scheme, use the following variation of the UNION statement to allow one section to be initialized and the remaining sections to be uninitialized.

Generally, an initialized section (one with raw data, such as .text) in a union must have its load allocation specified separately. However, one and only one initialized section in a union can be allocated at the union's run address. By listing it in the UNION statement with no load allocation at all, it will use the union's run address as its own load address. For example:

```
UNION run = FAST_MEM
{ .cinit .bss }
```

In this example, the .cinit section is an initialized section. It will be loaded into FAST_MEM at the run address of the union. In contrast, .bss is an uninitialized section. Its run address will also be that of the union.

12.5.7.4 Nesting UNIONs and GROUPs

The linker allows arbitrary nesting of GROUP and UNION statements with the SECTIONS directive. By nesting GROUP and UNION statements, you can express hierarchical overlays and groupings of sections. Nesting GROUP and UNION Statements shows how two overlays can be grouped together.

Nesting GROUP and UNION Statements

```
SECTIONS
{
   GROUP 0x1000 : run = FAST_MEM
   {
      UNION:
      {
         mysect1: load = SLOW_MEM
         mysect2: load = SLOW_MEM
      }
      UNION:
      {
         mysect3: load = SLOW_MEM
         mysect4: load = SLOW_MEM
      }
   }
}
```

For this example, the linker performs the following allocations:

- The four sections (mysect1, mysect2, mysect3, mysect4) are assigned unique, non-overlapping load addresses. This assignment is determined by the particular load allocations given for each section.
- Sections mysect1 and mysect2 are assigned the same run address in FAST_MEM.
- Sections mysect3 and mysect4 are assigned the same run address in FAST_MEM.
- The run addresses of mysect1/mysect2 and mysect3/mysect4 are allocated contiguously, as directed by the GROUP statement (subject to alignment and blocking restrictions).

To refer to groups and unions, linker diagnostic messages use the notation:

GROUP_*n* UNION_*n*

where *n* is a sequential number (beginning at 1) that represents the lexical ordering of the group or union in the linker control file without regard to nesting. Groups and unions each have their own counter.

Copyright © 2026 Texas Instruments Incorporated

12.5.7.5 Checking the Consistency of Allocators

The linker checks the consistency of load and run allocations specified for unions, groups, and sections. The following rules are used:

- Run allocations are only allowed for top-level sections, groups, or unions (sections, groups, or unions that are not nested under any other groups or unions). The linker uses the run address of the top-level structure to compute the run addresses of the components within groups and unions.
- The linker does not accept a load allocation for UNIONs.
- The linker does not accept a load allocation for uninitialized sections.
- In most cases, you must provide a load allocation for an initialized section. However, the linker does not accept a load allocation for an initialized section that is located within a group that already defines a load allocator.
- As a shortcut, you can specify a load allocation for an entire group, to determine the load allocations for every initialized section or subgroup nested within the group. However, a load allocation is accepted for an entire group only if all of the following conditions are true:
  - The group is initialized (that is, it has at least one initialized member).
  - The group is not nested inside another group that has a load allocator.
  - The group does not contain a union containing initialized sections.
- If the group contains a union with initialized sections, it is necessary to specify the load allocation for each initialized section nested within the group. Consider the following example:

```
SECTIONS
{
  GROUP: load = SLOW_MEM, run = SLOW_MEM
  {
   .text1:
   UNION:
   {
    .text2:
    .text3:
   }
  }
}
```

  The load allocator given for the group does not uniquely specify the load allocation for the elements within the union: .text2 and .text3. In this case, the linker issues a diagnostic message to request that these load allocations be specified explicitly.

12.5.7.6 Naming UNIONs and GROUPs

You can give a name to a UNION or GROUP by entering the name in parentheses after the declaration. For example:

```
GROUP(BSS_SYSMEM_STACK_GROUP)
{
   .bss    :{}
   .sysmem :{}
   .stack  :{}
} load=D_MEM,  run=D_MEM
```

The name you defined is used in diagnostics for easy identification of the problem LCF area. For example:

```
warning: LOAD placement ignored for "BSS_SYSMEM_STACK_GROUP": object is uninitialized
   UNION(TEXT_CINIT_UNION)
   {
      .const :{}load=D_MEM, table(table1)
      .pinit :{}load=D_MEM, table(table1)
   }run=P_MEM
warning:table(table1) operator ignored: table(table1) has already been applied to a section
in the "UNION(TEXT_CINIT_UNION)" in which ".pinit" is a descendant
```


Copyright © 2026 Texas Instruments Incorporated

12.5.8 Special Section Types (DSECT, COPY, NOLOAD, and NOINIT)

You can assign the following special types to output sections: DSECT, COPY, NOLOAD, and NOINIT. These types affect the way that the program is treated when it is linked and loaded. You can assign a type to a section by placing the type after the section definition. For example:

```
SECTIONS
   {
   sec1: load = 0x00002000, type =  DSECT  {f1.c.obj}
   sec2: load = 0x00004000, type =  COPY   {f2.c.obj}
   sec3: load = 0x00006000, type =  NOLOAD {f3.c.obj}
   sec4: load = 0x00008000, type =  NOINIT {f4.c.obj}
   }
```

- The DSECT type creates a dummy section with the following characteristics:
  - It is not included in the output section memory allocation. It takes up no memory and is not included in the memory map listing.
  - It can overlay other output sections, other DSECTs, and unconfigured memory.
  - Global symbols defined in a dummy section are relocated normally. They appear in the output module's symbol table with the same value they would have if the DSECT had actually been loaded. These symbols can be referenced by other input sections.
  - Undefined external symbols found in a DSECT cause specified archive libraries to be searched.
  - The section's contents, relocation information, and line number information are not placed in the output module.

  In the preceding example, none of the sections from f1.c.obj are allocated, but all the symbols are relocated as though the sections were linked at address 0x2000. The other sections can refer to any of the global symbols in sec1.
- A COPY section is similar to a DSECT section, except that its contents and associated information are written to the output module. The .cinit section that contains initialization tables for the C7000 C/C++ compiler has this attribute under the run-time initialization model.
- A NOLOAD section differs from a normal output section in one respect: the section's contents, relocation information, and line number information are not placed in the output module. The linker allocates space for the section, and it appears in the memory map listing.
- A NOINIT section is not C auto-initialized by the linker. It is your responsibility to initialize this section as needed.

12.5.9 Assigning Symbols at Link Time

Linker assignment statements allow you to define external (global) symbols and assign values to them at link time. You can use this feature to initialize a variable or pointer to an allocation-dependent value. See Section 12.6 for information about referring to linker symbols in C/C++ code.

12.5.9.1 Syntax of Assignment Statements

The syntax of assignment statements in the linker is similar to that of assignment statements in the C language:

symbol	=	*expression;*	assigns the value of expression to symbol
symbol	+ =	*expression;*	adds the value of expression to symbol
symbol	- =	*expression;*	subtracts the value of expression from symbol
symbol	* =	*expression;*	multiplies symbol by expression
symbol	/ =	*expression;*	divides symbol by expression

The symbol should be defined externally. If it is not, the linker defines a new symbol and enters it into the symbol table. The expression must follow the rules defined in Section 12.5.9.3. Assignment statements *must* terminate with a semicolon.

The linker processes assignment statements *after* it allocates all the output sections. Therefore, if an expression contains a symbol, the address used for that symbol reflects the symbol's address in the executable output file.

Copyright © 2026 Texas Instruments Incorporated

For example, suppose a program reads data from one of two tables identified by two external symbols, Table1 and Table2. The program uses the symbol cur_tab as the address of the current table. The cur_tab symbol must point to either Table1 or Table2. You can use a linker assignment statement to assign cur_tab at link time:

```
prog.c.obj          /* Input file                          */
cur_tab = Table1;  /* Assign cur_tab to one of the tables */
```

12.5.9.2 Assigning the SPC to a Symbol

A special symbol, denoted by a dot (.), represents the current value of the section program counter (SPC) during allocation. The SPC keeps track of the current location within a section. The **.** symbol can be used only in assignment statements within a SECTIONS directive because . is meaningful only during allocation and SECTIONS controls the allocation process. (See Section 12.5.5.)

The **.** symbol refers to the current run address, not the current load address, of the section.

A special type of assignment assigns a value to the **.** (dot) symbol. This adjusts the SPC within an output section and creates a hole between two input sections. Any value assigned to **.** to create a hole is relative to the beginning of the section, not to the address actually represented by the **.** symbol. Holes and assignments to **.** are described in Section 12.5.10.

12.5.9.3 Assignment Expressions

These rules apply to linker expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in Table 12-11.
- All numbers are treated as long (64-bit) integers.
- Constants are identified by the linker in the same way as by the assembler. That is, numbers are recognized as decimal unless they have a suffix (H or h for hexadecimal and Q or q for octal). C language prefixes are also recognized (0 for octal and 0x for hex). Hexadecimal constants must begin with a digit. No binary constants are allowed.
- Symbols within an expression have only the value of the symbol's *address*. No type-checking is performed.
- Linker expressions can be absolute or relocatable. If an expression contains *any* relocatable symbols (and 0 or more constants or absolute symbols), it is relocatable. Otherwise, the expression is absolute. If a symbol is assigned the value of a relocatable expression, it is relocatable; if it is assigned the value of an absolute expression, it is absolute.

The linker supports the C language operators listed in Table 12-11 in order of precedence. Operators in the same group have the same precedence. Besides the operators listed in Table 12-11, the linker also has an align operator that allows a symbol to be aligned on an n-byte boundary within an output section (n is a power of 2). For example, the following expression aligns the SPC within the current section on the next 16-byte boundary. Because the align operator is a function of the current SPC, it can be used only in the same context as . —that is, within a SECTIONS directive.

```
    . = align(16);
```

Table 12-11. Groups of Operators Used in Expressions (Precedence)

Group 1 (Highest Precedence)		Group 6	
!	Logical NOT	&	Bitwise AND
~	Bitwise NOT		
-	Negation		
Group 2		**Group 7**	
*	Multiplication	|	Bitwise OR
/	Division		
%	Modulus		
Group 3		**Group 8**	
+	Addition	&&	Logical AND
-	Subtraction		

Copyright © 2026 Texas Instruments Incorporated

Table 12-11. Groups of Operators Used in Expressions (Precedence) (continued)

Group 1 (Highest Precedence)		Group 6			
Group 4		**Group 9**			
>>	Arithmetic right shift	||	Logical OR		
<<	Arithmetic left shift				
Group 5		**Group 10 (Lowest Precedence)**			
==	Equal to	=	Assignment		
! =	Not equal to	+ =	A + = B	is equivalent to	A = A + B
>	Greater than	- =	A - = B	is equivalent to	A = A - B
<	Less than	* =	A * = B	is equivalent to	A = A * B
< =	Less than or equal to	/ =	A / = B	is equivalent to	A = A / B
> =	Greater than or equal to				

12.5.9.4 Symbols Automatically Defined by the Linker

.text is assigned the first address of the .text output section. (It marks the *beginning* of executable code.)

etext is assigned the first address following the .text output section. (It marks the *end* of executable code.)

.data is assigned the first address of the .data output section. (It marks the *beginning* of initialized data tables.)

edata is assigned the first address following the .data output section. (It marks the *end* of initialized data tables.)

.bss is assigned the first address of the .bss output section. (It marks the *beginning* of uninitialized data.)

end is assigned the first address following the .bss output section. (It marks the *end* of uninitialized data.)

The linker automatically defines the following symbols for C/C++ support when the --ram_model or --rom_model option is used.

__TI_STACK_SIZE is assigned the size of the .stack section.

__TI_STACK_END is assigned the end of the .stack section.

__TI_SYSMEM_SIZE is assigned the size of the .sysmem section.

__TI_STATIC_BASE is assigned the value to be loaded into the data pointer register (DP) at boot time. This is typically the start of the first section containing a definition of a symbol that is referenced via near-DP addressing.

See Section 12.6 for information about referring to linker symbols in C/C++ code.

12.5.9.5 Assigning Exact Start, End, and Size Values of a Section to a Symbol

The code generation tools currently support the ability to load program code in one area of (slow) memory and run it in another (faster) area. This is done by specifying separate load and run addresses for an output section or group in the linker command file. Then execute a sequence of instructions that moves the program code from its load area to its run area before it is needed.

There are several responsibilities that a programmer must take on when setting up a system with this feature. One of these responsibilities is to determine the size and run-time address of the program code to be moved. The current mechanisms to do this involve use of the .label directives in the copying code.

This method of specifying the size and load address of the program code has limitations. While it works fine for an individual input section that is contained entirely within one source file, this method becomes more complicated if the program code is spread over several source files or if the programmer wants to copy an entire output section from load space to run space.

Another problem with this method is that it does not account for the possibility that the section being moved may have an associated far call trampoline section that needs to be moved with it.

Copyright © 2026 Texas Instruments Incorporated

12.5.9.6 Why the Dot Operator Does Not Always Work

The dot operator (.) is used to define symbols at link-time with a particular address inside of an output section. It is interpreted like a PC. Whatever the current offset within the current section is, that is the value associated with the dot. Consider an output section specification within a SECTIONS directive:

```
outsect:
{
        s1.c.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.c.obj(.text)
        end_of_s2 = .;
}
```

This statement creates three symbols:

- end_of_s1—the end address of .text in s1.c.obj
- start_of_s2—the start address of .text in s2.c.obj
- end_of_s2—the end address of .text in s2.c.obj

Suppose there is padding between s1.c.obj and s2.c.obj created as a result of alignment. Then start_of_s2 is not really the start address of the .text section in s2.c.obj, but it is the address before the padding needed to align the .text section in s2.c.obj. This is due to the linker's interpretation of the dot operator as the current PC. It is also true because the dot operator is evaluated independently of the input sections around it.

Another potential problem in the above example is that end_of_s2 may not account for any padding that was required at the end of the output section. You cannot reliably use end_of_s2 as the end address of the output section. One way to get around this problem is to create a dummy section immediately after the output section in question. For example:

```
GROUP
{
   outsect:
   {
      start_of_outsect = .;
      ...
   }
      dummy: { size_of_outsect = . - start_of_outsect; }
}
```

12.5.9.7 Address and Dimension Operators

Six operators allow you to define symbols for load-time and run-time addresses and sizes:

LOAD_START(*sym***)** **START(** *sym***)**	Defines *sym* with the load-time start address of related allocation unit
LOAD_END(*sym***)** **END(** *sym***)**	Defines *sym* with the load-time end address of related allocation unit
LOAD_SIZE(*sym***)** **SIZE(** *sym***)**	Defines *sym* with the load-time size of related allocation unit
RUN_START(*sym***)**	Defines *sym* with the run-time start address of related allocation unit
RUN_END(*sym***)**	Defines *sym* with the run-time end address of related allocation unit
RUN_SIZE(*sym***)**	Defines *sym* with the run-time size of related allocation unit
LAST(*sym***)**	Defines *sym* with the run-time address of the last allocated byte in the related memory range.

Note

Linker Command File Operator Equivalencies: LOAD_START() and START() are equivalent, as are LOAD_END()/END() and LOAD_SIZE()/SIZE(). The LOAD names are recommended for clarity.

Copyright © 2026 Texas Instruments Incorporated

These address and dimension operators can be associated with several different kinds of allocation units, including input items, output sections, GROUPs, and UNIONs. The following sections provide some examples of how the operators can be used in each case.

These symbols defined by the linker can be accessed at runtime using the _symval operator, which is essentially a cast operation. For example, suppose your linker command file contains the following:

```
.text: RUN_START(text_run_start), RUN_SIZE(text_run_size) { *(.text) }
```

Your C program can access these symbols as follows:

```
extern char text_run_start, text_run_size;
printf(".text load start is %lx\n", _symval(&text_run_start));
printf(".text load size is %lx\n",  _symval(&text_run_size));
```

See Section 12.6 for more information about referring to linker symbols in C/C++ code.

12.5.9.7.1 Input Items

Consider an output section specification within a SECTIONS directive:

```
outsect:
{
        s1.c.obj(.text)
        end_of_s1   = .;
        start_of_s2 = .;
        s2.c.obj(.text)
        end_of_s2 = .;
}
```

This can be rewritten using the START and END operators as follows:

```
outsect:
{
        s1.c.obj(.text) { END(end_of_s1) }
        s2.c.obj(.text) { START(start_of_s2), END(end_of_s2) }
}
```

The values of end_of_s1 and end_of_s2 will be the same as if you had used the dot operator in the original example, but start_of_s2 would be defined after any necessary padding that needs to be added between the two .text sections. Remember that the dot operator would cause start_of_s2 to be defined before any necessary padding is inserted between the two input sections.

The syntax for using these operators in association with input sections calls for braces { } to enclose the operator list. The operators in the list are applied to the input item that occurs immediately before the list.

Copyright © 2026 Texas Instruments Incorporated

12.5.9.7.2 Output Section

The START, END, and SIZE operators can also be associated with an output section. Here is an example:

```
outsect: START(start_of_outsect), SIZE(size_of_outsect)
{
        <list of input items>
}
```

In this case, the SIZE operator defines size_of_outsect to incorporate any padding that is required in the output section to conform to any alignment requirements that are imposed.

The syntax for specifying the operators with an output section does not require braces to enclose the operator list. The operator list is simply included as part of the allocation specification for an output section.

12.5.9.7.3 GROUPs

Here is another use of the START and SIZE operators in the context of a GROUP specification:

```
GROUP
{
        outsect1: { ... }
        outsect2: { ... }
} load = ROM, run = RAM, START(group_start), SIZE(group_size);
```

This can be useful if the whole GROUP is to be loaded in one location and run in another. The copying code can use group_start and group_size as parameters for where to copy from and how much is to be copied.

12.5.9.7.4 UNIONs

The RUN_SIZE and LOAD_SIZE operators provide a mechanism to distinguish between the size of a UNION's load space and the size of the space where its constituents are going to be copied before they are run. Here is an example:

```
UNION: run = RAM, LOAD_START(union_load_addr),
       LOAD_SIZE(union_ld_sz), RUN_SIZE(union_run_sz)
{
        .text1: load = ROM, SIZE(text1_size) { f1.c.obj(.text) }
        .text2: load = ROM, SIZE(text2_size) { f2.c.obj(.text) }
}
```

Here union_ld_sz is going to be equal to the sum of the sizes of all output sections placed in the union. The union_run_sz value is equivalent to the largest output section in the union. Both of these symbols incorporate any padding due to blocking or alignment requirements.

12.5.9.8 LAST Operator

The LAST operator is similar to the START and END operators that were described previously. However, LAST applies to a memory range rather than to a section. You can use it in a MEMORY directive to define a symbol that can be used at run-time to learn how much memory was allocated when linking the program. See Section 12.5.4.2 for syntax details.

For example, a memory range might be defined as follows:

```
D_MEM : org = 0x20000020 len = 0x20000000 LAST(dmem_end)
```

Your C program can then access this symbol at runtime using the _symval operator. For example:

```
extern char dmem_end;
printf("End of D_MEM memory is %lx\n", _symval(&dmem_end));
```

See Section 12.6 for more information about referring to linker symbols in C/C++ code.

Copyright © 2026 Texas Instruments Incorporated

12.5.10 Creating and Filling Holes

The linker provides you with the ability to create areas *within output sections* that have nothing linked into them. These areas are called *holes*. In special cases, uninitialized sections can also be treated as holes. This section describes how the linker handles holes and how you can fill holes (and uninitialized sections) with values.

12.5.10.1 Initialized and Uninitialized Sections

There are two rules to remember about the contents of output sections. An output section contains either:

- Raw data for the *entire* section
- *No* raw data

A section that has raw data is referred to as *initialized*. This means that the object file contains the actual memory image contents of the section. When the section is loaded, this image is loaded into memory at the section's specified starting address. The .text and .data sections *always* have raw data if anything was assembled into them.

By default, the .bss section and sections defined with the .usect directive have no raw data (they are *uninitialized*). They occupy space in the memory map but have no actual contents. Uninitialized sections typically reserve space in fast external memory for variables. In the object file, an uninitialized section has a normal section header and can have symbols defined in it; no memory image, however, is stored in the section.

12.5.10.2 Creating Holes

You can create a hole in an initialized output section. A hole is created when you force the linker to leave extra space between input sections within an output section. When such a hole is created, *the linker must supply raw data for the hole.*

Holes can be created only *within* output sections. Space can exist *between* output sections, but such space is not a hole. To fill the space between output sections, see Section 12.5.4.2.

To create a hole in an output section, you must use a special type of linker assignment statement within an output section definition. The assignment statement modifies the SPC (denoted by **.**) by adding to it, assigning a greater value to it, or aligning it on an address boundary. The operators, expressions, and syntaxes of assignment statements are described in Section 12.5.9.

The following example uses assignment statements to create holes in output sections:

```
SECTIONS
{
   outsect:
   {
      file1.c.obj(.text)
     . += 0x0100    /* Create a hole with size 0x0100 */
      file2.c.obj(.text)
    . = align(16);  /* Create a hole to align the SPC */
      file3.c.obj(.text)
    }
}
```

The output section outsect is built as follows:

1. The .text section from file1.c.obj is linked in.
2. The linker creates a 256-byte hole.
3. The .text section from file2.c.obj is linked in after the hole.
4. The linker creates another hole by aligning the SPC on a 16-byte boundary.
5. Finally, the .text section from file3.c.obj is linked in.

All values assigned to the **.** symbol within a section refer to the *relative address within the section.* The linker handles assignments to the **.** symbol as if the section started at address 0 (even if you have specified a binding address). Consider the statement **.** = align(16) in the example. This statement effectively aligns the

Copyright © 2026 Texas Instruments Incorporated

file3.c.obj .text section to start on a 16-byte boundary within outsect. If outsect is ultimately allocated to start on an address that is not aligned, the file3.c.obj .text section will not be aligned either.

The **.** symbol refers to the current run address, not the current load address, of the section.

Expressions that decrement the **.** symbol are illegal. For example, it is invalid to use the -= operator in an assignment to the **.** symbol. The most common operators used in assignments to the **.** symbol are += and align.

If an output section contains all input sections of a certain type (such as .text), you can use the following statements to create a hole at the beginning or end of the output section.

```
    .text:   {  .+= 0x0100; }     /* Hole at the beginning */
    .data:   {  *(.data)
                . += 0x0100; }    /* Hole at the end       */
```

Another way to create a hole in an output section is to combine an uninitialized section with an initialized section to form a single output section. *In this case, the linker treats the uninitialized section as a hole and supplies data for it.* The following example illustrates this method:

```
SECTIONS
{
   outsect:
   {
      file1.c.obj(.text)
     file1.c.obj(.bss)           /* This becomes a hole */
   }
}
```

Because the .text section has raw data, all of outsect must also contain raw data. Therefore, the uninitialized .bss section becomes a hole.

Uninitialized sections become holes only when they are combined with initialized sections. If several uninitialized sections are linked together, the resulting output section is also uninitialized.

12.5.10.3 Filling Holes

If there is a hole in an initialized output section, the linker supplies raw data to fill it. The linker fills holes with a 64-bit fill value replicated through memory until it fills the hole. The linker determines the fill value as follows:

1. If a hole is formed by combining an uninitialized section with an initialized section, you can specify a fill value for the uninitialized section. Follow the section name with an = sign and a 64-bit constant. For example:

   ```
   SECTIONS
   {  outsect:
      {
         file1.c.obj(.text)
         file2.c.obj(.bss)= 0xFF00FF00  /* Fill this hole with 0xFF00FF00 */
      }
   }
   ```

2. You can also specify a fill value for all the holes in an output section by supplying the fill value after the section definition:

   ```
   SECTIONS
   {  outsect:fill = 0xFF00FF00       /* Fills holes with 0xFF00FF00 */
      {
         . += 0x0010;                 /* This creates a hole         */
         file1.c.obj(.text)
         file1.c.obj(.bss)              /* This creates another hole   */
      }
   }
   ```


Copyright © 2026 Texas Instruments Incorporated

3. If you do not specify an initialization value for a hole, the linker fills the hole with the value specified with the --fill_value option (see Section 12.4.12). For example, suppose the command file link.cmd contains the following SECTIONS directive:

```
SECTIONS { .text: { .= 0x0100; } /* Create a 100 word hole */ }
```

   Now invoke the linker with the --fill_value option:

```
cl7x --run_linker --fill_value=0xFFFFFFFFFFFFFFFF link.cmd
```

   This fills the hole with 0xFFFFFFFF.
4. If invoke the linker without the --fill_value option or specify a fill value, the linker fills holes with 0s.

Whenever a hole is created and filled in an initialized output section, the hole is identified in the link map along with the value the linker uses to fill it.

12.5.10.4 Explicit Initialization of Uninitialized Sections

You can force the linker to initialize an uninitialized section by specifying an explicit fill value for it in the SECTIONS directive. This causes the entire section to have raw data (the fill value). For example:

```
SECTIONS
{
   .bss: fill = 0x1234123412341234 /* Fills .bss with 0x1234123412341234 */
}
```

Note

Filling Sections: Filling a section (even with 0s) generates raw data for the entire section in the output file, so your output file will be large if you specify fill values for large sections or holes.

Copyright © 2026 Texas Instruments Incorporated

12.6 Linker Symbols

C/C++ source code may need to refer to symbols that are defined by the linker, not in C/C++ source code. For linker-defined symbols that act like a function or array, you can often refer to such symbols in a straightforward way in C/C++ code. For other purposes in C/C++ code, you typically need to use other techniques, such as the _symval operator, to access linker-defined symbols. These techniques are described in the subsections that follow.

12.6.1 Linker-Defined Functions and Arrays

In most cases, you can access linker-defined functions in the same way as C/C++ functions. Provide an extern declaration (prototype) for that function, and access the function normally:

```
extern int linker_defined_function(void);
printf("value is %d\n", linker_defined_function());
```

In most cases, you can access linker-defined arrays in the same way as C/C++ arrays. Provide an extern declaration for the array (which can omit the first dimension), and access the array normally:

```
extern int linker_defined_data[][10][10];
printf("value is %d\n", linker_defined_data[2][3][4]);
```

If you receive a relocation error because the function or array is outside the normal address range, use the _symval operator as described in Section 12.6.4.

12.6.2 Linker-Defined Integer Values

To access linker symbols that represent integer values, use the _symval built-in operator, which is essentially a cast operation.

For example, the linker symbol __TI_STACK_SIZE evaluates to a plain integer. To get the symbol's value as an integer in C/C++ code, use the following syntax:

```
extern void __TI_STACK_SIZE;
size_t get_stack_size() { return _symval(&__TI_STACK_SIZE); }
```

The type in such extern declarations does not matter, because only the address of the symbol is needed. In strict ANSI mode, you cannot declare this variable with a type of void, so use unsigned char instead.

Note

Do not attempt to use the __TI_STACK_SIZE symbol by itself in C/C++. The symbol's value is undefined, so the symbol's address is likely to be an invalid memory address.

To understand why the _symval operator is needed to access linker-defined integer values, see Section 12.6.4 for information about how linker symbols differ from C identifiers.

12.6.3 Linker-Defined Addresses

To access linker-defined symbols that represent addresses, use the _symval built-in operator to get the symbol's value as a pointer value.

For example, the linker symbol __TI_STACK_END evaluates to an address. To get the symbol's value as a pointer value in C/C++ code, use the following syntax:

```
extern void __TI_STACK_END;
void *get_stack_end() { return (void*)_symval(&__TI_STACK_END); }
```


Copyright © 2026 Texas Instruments Incorporated

Although the linker symbol __TI_STACK_END is an address, and thus looks a lot like a C/C++ pointer value, you cannot simply define the C/C++ variable __TI_STACK_END as a pointer variable and omit taking the symbol's address. See Section 12.6.4 for details. The following example is *incorrect*.

```
extern void * __TI_STACK_END;
void *get_stack_end() { return __TI_STACK_END; }  // wrong, missing &
```

12.6.4 More About the _symval Operator

When you declare a variable such as `int x = 1234;` in C/C++, an object with the identifier (name) “x” is created at address &x with the contents 1234. In addition, a related linker symbol named x is created. That linker symbol represents just the address, not the object itself. References to the linker symbol x result in the linker symbol’s value, which is the address. However, references to the C/C++ identifier x result in the contents 1234. If you want the address of this C/C++ identifier, you need to use &x. Thus, the C/C++ expression &x has the same value as the linker expression x, although the linker symbol does not have an associated type.

Suppose a linker-defined symbol represents an integer rather than an address, such as __TI_STACK_SIZE. There is no way to refer to an integer linker symbol directly in the compiler, so we use a trick. First, pretend this linker symbol represents an address. Declare a fake variable with the same name in the C/C++ code. Now, refer to &__TI_STACK_SIZE, an expression that has the same value as the linker symbol __TI_STACK_SIZE. The value does have the wrong type, which you can change with a cast as follows:

```
extern unsigned char __TI_STACK_SIZE;
size_t stack_size = (size_t) &__TI_STACK_SIZE;
```

Leaving out _symval as shown in the above example works most of the time, but not always. In some cases, a pointer value is not adequate to represent a linker symbol value. For example, some targets have a 16-bit address space and thus 16-bit pointers. TI linker symbols are 64 bits, so a linker symbol can have a value that is larger than can be represented by a target pointer. In such cases, the expression &v reflects only the lower 16 bits of the actual value of the linker symbol “v”. To get around this problem, use the _symval built-in operator, which causes all of the value bits of the linker symbol to be copied:

```
extern void v;
unsigned long value = (unsigned long)_symval(&v);
```

For every kind of linker symbol, use this pattern:

```
extern void name;
desired_type name = (desired_type)_symval(&name);
```

For example,

```
extern void farfunc;

void foo()
{
    void (*func)(void) = (void (*)(void))_symval(&farfunc);
    func();
}
```

On C7000 devices, linker symbols that represent integers or that for some other reason do not represent valid addresses, cannot always be represented with a pointer value. Use _symval in such cases to make sure you get the right value.

12.6.5 Using _symval, PC-Relative Addressing, and Far Data

On C7000 devices, most symbol references are handled in a position-independent manner through a PC-relative relocation. This sort of relocation can reach only +/- 2GB away from the current PC value. However, C7000 devices can have objects beyond this +/- 2GB reach, in which case a PC-relative relocation would overflow and a linker error would occur.

Copyright © 2026 Texas Instruments Incorporated

To access such far-away objects, use the _symval operator to get the full address. Using _symval forces the linker to use an absolute relocation, which uses the full address.

12.6.6 Weak Symbols

Weak symbols are symbols that may or may not be defined.

See Section 8.5.2 for details about how weak symbols are processed by the linker.

12.6.6.1 Weak Symbol References

Weak symbol references may or may not have a definition after the final link is performed. If a symbol is undefined, its address is considered to be 0. C/C++ code must check the address of weak references to make sure the value is not 0 before attempting to use the contents of that variable.

```
extern __attribute__((weak)) unsigned char * foo;
if (&foo != 0)
    *foo = 1;
```

If the linker symbol corresponding to foo might not have a valid address (for instance, because the symbol contains an integer value instead of an address) or might be beyond the 2 GB reach of PC-relative addressing, use the _symval built-in operator as follows:

```
extern __attribute__((weak)) unsigned char * foo;
if (_symval(&foo) != 0)
    *foo = 1;
```

12.6.6.2 Weak Symbol Definitions

A weak symbol definition is a valid definition, but the definition is discarded in favor of a non-weak definition if such a definition is found at link time. You can define weak symbols C/C++ or in the linker command file.

In C/C++, define a weak symbol as follows:

```
__attribute__((weak)) int bar;
```

In a linker command file, use an assignment expression outside a MEMORY or SECTIONS directive to define a linker-defined symbol. To define a weak symbol in a linker command file, use the "weak" operator in an assignment expression to designate that the symbol as eligible for removal from the output file's symbol table if the symbol is not referenced. For example, you can define "ext_addr_sym" as follows:

```
weak(ext_addr_sym) = 0x12345678;
```

When the linker command file is used to perform the final link, "ext_addr_sym" is presented to the linker as a weak absolute symbol. This symbol is not included in the resulting output file if the symbol is not referenced.

12.6.7 Resolving Symbols with Object Libraries

An object library is a partitioned archive file that contains object files as members. Usually, a group of related modules are grouped together into a library. When you specify an object library as linker input, the linker includes any members of the library that define existing unresolved symbol references. You can use the archiver to build and maintain libraries. Section 10.1 contains more information about the archiver.

Using object libraries can reduce link time and the size of the executable module. Normally, if an object file that contains a function is specified at link time, the file is linked whether the function is used or not; however, if that same function is placed in an archive library, the file is included only if the function is referenced.

The order in which libraries are specified is important, because the linker includes only those members that resolve symbols that are undefined at the time the library is searched. The same library can be specified as often as necessary; it is searched each time it is included. Alternatively, you can use the --reread_libs option to reread libraries until no more references can be resolved (see Section 12.4.15.3). A library has a table that lists

Copyright © 2026 Texas Instruments Incorporated

all external symbols defined in the library; the linker searches through the table until it determines that it cannot use the library to resolve any more references.

The following examples link several files and libraries, using these assumptions:

- Input files f1.c.obj and f2.c.obj both reference an external function named *clrscr*.
- Input file f1.c.obj references the symbol *origin*.
- Input file f2.c.obj references the symbol *fillclr*.
- Member 0 of library libc.lib contains a definition of *origin*.
- Member 3 of library liba.lib contains a definition of *fillclr*.
- Member 1 of both libraries defines *clrscr*.

If you enter:

```
cl7x --run_linker f1.c.obj f2.c.obj liba.lib libc.lib
```

then:

- Member 1 of liba.lib satisfies the f1.c.obj and f2.c.obj references to *clrscr* because the library is searched and the definition of *clrscr* is found.
- Member 0 of libc.lib satisfies the reference to *origin*.
- Member 3 of liba.lib satisfies the reference to *fillclr*.

If, however, you enter:

```
cl7x --run_linker f1.c.obj f2.c.obj libc.lib liba.lib
```

then the references to *clrscr* are satisfied by member 1 of libc.lib.

If none of the linked files reference symbols defined in a library, you can use the --undef_sym option to force the linker to include a library member. (See Section 12.4.30.) The next example creates an undefined symbol rout1 in the linker's global symbol table:

```
cl7x --run_linker --undef_sym=rout1 libc.lib
```

If any member of libc.lib defines rout1, the linker includes that member.

Library members are allocated according to the SECTIONS directive default allocation algorithm; see Section 12.5.5.

Section 12.4.15 describes methods for specifying directories that contain object libraries.

Copyright © 2026 Texas Instruments Incorporated

12.7 Default Placement Algorithm

The MEMORY and SECTIONS directives provide flexible ways to build, combine, and allocate sections. However, any memory locations or sections you choose *not* to specify must still be handled by the linker. The linker uses algorithms to build and allocate sections in coordination with any specifications you do supply.

If you do not use the MEMORY and SECTIONS directives, the linker allocates sections starting at a memory address near zero. It begins by placing .text sections and then places the various data sections after that.

See Section 8.4.1 for information about default memory allocation.

All .text input sections are concatenated to form a .text output section in the executable output file, and all .data input sections are combined to form a .data output section.

If you use a SECTIONS directive, allocation is performed according to the rules specified by the SECTIONS directive and the general algorithm described next in Section 12.7.1.

12.7.1 How the Allocation Algorithm Creates Output Sections

An output section can be formed in one of two ways:

Method 1	As the result of a SECTIONS directive definition
Method 2	By combining input sections with the same name into an output section that is not defined in a SECTIONS directive

If an output section is formed as a result of a SECTIONS directive, this definition completely determines the section's contents. (See Section 12.5.5 for examples of how to define an output section's content.)

If an output section is formed by combining input sections not specified by a SECTIONS directive, the linker combines all such input sections that have the same name into an output section with that name. For example, suppose the files f1.c.obj and f2.c.obj both contain named sections called Vectors and that the SECTIONS directive does not define an output section for them. The linker combines the two Vectors sections from the input files into a single output section named Vectors, allocates it into memory, and includes it in the output file.

By default, the linker does not display a message when it creates an output section that is not defined in the SECTIONS directive. You can use the --warn_sections linker option (see Section 12.4.31) to cause the linker to display a message when it creates a new output section.

After the linker determines the composition of all output sections, it must allocate them into configured memory. The MEMORY directive specifies which portions of memory are configured. If there is no MEMORY directive, the linker uses the default configuration. (See Section 12.5.4 for more information on configuring memory.)

12.7.2 Reducing Memory Fragmentation

The linker's allocation algorithm attempts to minimize memory fragmentation. This allows memory to be used more efficiently and increases the probability that your program will fit into memory. The algorithm comprises these steps:

1. Each output section for which you supply a specific binding address is placed in memory at that address.
2. Each output section that is included in a specific, named memory range or that has memory attribute restrictions is allocated. Each output section is placed into the first available space within the named area, considering alignment where necessary.
3. Any remaining sections are allocated in the order in which they are defined. Sections not defined in a SECTIONS directive are allocated in the order in which they are encountered. Each output section is placed into the first available memory space, considering alignment where necessary.

Copyright © 2026 Texas Instruments Incorporated

12.8 Using Linker-Generated Copy Tables

The linker supports extensions to the linker command file syntax that enable the following:

- Make it easier for you to copy objects from load-space to run-space at boot time
- Make it easier for you to manage memory overlays at run time
- Allow you to split GROUPs and output sections that have separate load and run addresses

For an introduction to copy tables and their use, see Section 9.3.3.

12.8.1 Using Copy Tables for Boot Loading

In some embedded applications, there is a need to copy or download code and/or data from one location to another at boot time before the application actually begins its main execution thread. For example, an application may have its code and/or data in FLASH memory and need to copy it into on-chip memory before the application begins execution.

One way to develop such an application is to create a copy table that contains three elements for each block of code or data that needs to be moved from FLASH to on-chip memory at boot time:

- The load address
- The run address
- The size

The process you follow to develop such an application might look like this:

1. Build the application to produce a .map file that contains the load and run addresses of each section that has a separate load and run placement.
2. Edit the copy table (used by the boot loader) to correct the load and run addresses as well as the size of each block of code or data that needs to be moved at boot time.
3. Build the application again, incorporating the updated copy table.
4. Run the application.

This process puts a heavy burden on you to maintain the copy table (by hand, no less). Each time a piece of code or data is added or removed from the application, you must repeat the process in order to keep the contents of the copy table up to date.

12.8.2 Using Built-in Link Operators in Copy Tables

You can avoid some of this maintenance burden by using the LOAD_START(), RUN_START(), and SIZE() operators that are already part of the linker command file syntax. For example, instead of building the application to generate a .map file, the linker command file can be annotated:

```
SECTIONS
{
   .flashcode: { app_tasks.c.obj(.text) }
      load = FLASH, run = PMEM,
      LOAD_START(_flash_code_ld_start),
      RUN_START(_flash_code_rn_start),
      SIZE(_flash_code_size)
   ...
}
```

In this example, the LOAD_START(), RUN_START(), and SIZE() operators instruct the linker to create three symbols:

Symbol	Description
_flash_code_ld_start	Load address of .flashcode section
_flash_code_rn_start	Run address of .flashcode section
_flash_code_size	Size of .flashcode section

Copyright © 2026 Texas Instruments Incorporated

These symbols can then be referenced from the copy table. The actual data in the copy table will be updated automatically each time the application is linked. This approach removes step 1 of the process described in Section 12.8.1.

While maintenance of the copy table is reduced markedly, you must still carry the burden of keeping the copy table contents in sync with the symbols that are defined in the linker command file. Ideally, the linker would generate the boot copy table automatically. This would avoid having to build the application twice *and* free you from having to explicitly manage the contents of the boot copy table.

For more information on the LOAD_START(), RUN_START(), and SIZE() operators, see Section 12.5.9.7.

12.8.3 Overlay Management Example

Consider an application that contains a memory overlay that must be managed at run time. The memory overlay is defined using a UNION in the linker command file as illustrated in Using a UNION for Memory Overlay:

Using a UNION for Memory Overlay

```
SECTIONS
{
   ...
   UNION
   {
      GROUP
      {
         .task1: { task1.c.obj(.text) }
         .task2: { task2.c.obj(.text) }
      } load = ROM, LOAD_START(_task12_load_start), SIZE(_task12_size)
      GROUP
      {
         .task3: { task3.c.obj(.text) }
         .task4: { task4.c.obj(.text) }
      } load = ROM, LOAD_START(_task34_load_start), SIZE(_task_34_size)
   } run = RAM, RUN_START(_task_run_start)
 ...
}
```

The application must manage the contents of the memory overlay at run time. That is, whenever any services from .task1 or .task2 are needed, the application must first ensure that .task1 and .task2 are resident in the memory overlay. Similarly for .task3 and .task4.

To affect a copy of .task1 and .task2 from ROM to RAM at run time, the application must first gain access to the load address of the tasks (_task12_load_start), the run address (_task_run_start), and the size (_task12_size). Then this information is used to perform the actual code copy.

Copyright © 2026 Texas Instruments Incorporated

12.8.4 Generating Copy Tables With the table() Operator

The linker supports extensions to the linker command file syntax that enable you to do the following:

- Identify any object components that may need to be copied from load space to run space at some point during the run of an application
- Instruct the linker to automatically generate a copy table that contains (at least) the load address, run address, and size of the component that needs to be copied
- Instruct the linker to generate a symbol specified by you that provides the address of a linker-generated copy table. For instance, Using a UNION for Memory Overlay can be written as shown in the following example:

Produce Address for Linker Generated Copy Table

```
SECTIONS
{
   ...
   UNION
   {
      GROUP
      {
         .task1: { task1.c.obj(.text) }
         .task2: { task2.c.obj(.text) }
      } load = ROM, table(_task12_copy_table)
      GROUP
      {
         .task3: { task3.c.obj(.text) }
         .task4: { task4.c.obj(.text) }
      } load = ROM, table(_task34_copy_table)

   } run = RAM
   ...
}
```

Using the SECTIONS directive from the example above in the linker command file, the linker generates two copy tables named: _task12_copy_table and _task34_copy_table. Each copy table provides the load address, run address, and size of the GROUP that is associated with the copy table. This information is accessible from application source code using the linker-generated symbols, _task12_copy_table and _task34_copy_table, which provide the addresses of the two copy tables, respectively.

Using this method, you need not worry about the creation or maintenance of a copy table. You can reference the address of any copy table generated by the linker in C/C++ , passing that value to a general purpose copy routine, which will process the copy table and affect the actual copy.

12.8.4.1 The table() Operator

You can use the table() operator to instruct the linker to produce a copy table. A table() operator can be applied to an output section, a GROUP, or a UNION member. The copy table generated for a particular table() specification can be accessed through a symbol specified by you that is provided as an argument to the table() operator. The linker creates a symbol with this name and assigns it the address of the copy table as the value of the symbol. The copy table can then be accessed from the application using the linker-generated symbol.

Each table() specification you apply to members of a given UNION must contain a unique name. If a table() operator is applied to a GROUP, then none of that GROUP's members may be marked with a table() specification. The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the table() specification. The linker does not generate a copy table for erroneous table() operator specifications.

Copy tables can be generated automatically; see Section 12.8.4. The table operator can be used with compression; see Section 12.8.5.

Copyright © 2026 Texas Instruments Incorporated

12.8.4.2 Boot-Time Copy Tables

The linker supports a special copy table name, BINIT (or binit), that you can use to create a boot-time copy table. This table is handled before the .cinit section is used to initialize variables at startup. For example, the linker command file for the boot-loaded application described in Section 12.8.2 can be rewritten as follows:

```
SECTIONS
{
   .flashcode: { app_tasks.c.obj(.text) }
   load = FLASH, run = PMEM,
      table(BINIT)
   ...
}
```

For this example, the linker creates a copy table that can be accessed through a special linker-generated symbol, __binit__, which contains the list of all object components that need to be copied from their load location to their run location at boot-time. If a linker command file does not contain any uses of table(BINIT), then the __binit__ symbol is given a value of -1 to indicate that a boot-time copy table does not exist for a particular application.

You can apply the table(BINIT) specification to an output section, GROUP, or UNION member. If used in the context of a UNION, only one member of the UNION can be designated with table(BINIT). If applied to a GROUP, then none of that GROUP's members may be marked with table(BINIT).The linker detects violations of these rules and reports them as warnings, ignoring each offending use of the table(BINIT) specification.

12.8.4.3 Using the table() Operator to Manage Object Components

If you have several pieces of code that need to be managed together, then you can apply the same table() operator to several different object components. In addition, if you want to manage a particular object component in multiple ways, you can apply more than one table() operator to it. Consider the linker command file excerpt in the following example:

Linker Command File to Manage Object Components

```
SECTIONS
{
   UNION
   {
      .first: { a1.c.obj(.text), b1.c.obj(.text), c1.c.obj(.text) }
            load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)
      .second: { a2.c.obj(.text), b2.c.obj(.text) }
            load = EMEM, run = PMEM, table(_second_ctbl)
   }
   .extra: load = EMEM, run = PMEM, table(BINIT)
   ...
}
```

In this example, the output sections .first and .extra are copied from external memory (EMEM) into program memory (PMEM) at boot time while processing the BINIT copy table. After the application has started executing its main thread, it can then manage the contents of the overlay using the two overlay copy tables named: _first_ctbl and _second_ctbl.

12.8.4.4 Linker-Generated Copy Table Sections and Symbols

The linker creates and allocates a separate input section for each copy table that it generates. Each copy table symbol is defined with the address value of the input section that contains the corresponding copy table.

The linker generates a unique name for each overlay copy table input section. For example, table(_first_ctbl) would place the copy table for the .first section into an input section called .ovly:_first_ctbl. The linker creates a single input section, .binit, to contain the entire boot-time copy table.

The following example illustrates how you can control the placement of the linker-generated copy table sections using the input section names in the linker command file.

Copyright © 2026 Texas Instruments Incorporated

Controlling the Placement of the Linker-Generated Copy Table Sections

```
SECTIONS
{
   UNION
   {
      .first: { a1.c.obj(.text), b1.c.obj(.text), c1.c.obj(.text) }
              load = EMEM, run = PMEM, table(BINIT), table(_first_ctbl)
      .second: { a2.c.obj(.text), b2.c.obj(.text) }
              load = EMEM, run = PMEM, table(_second_ctbl)
   }
   .extra: load = EMEM, run = PMEM, table(BINIT)
   ...
   .ovly: { } > BMEM
   .binit: { } > BMEM
}
```

For the linker command file in this example, the boot-time copy table is generated into a .binit input section, which is collected into the .binit output section, which is mapped to an address in the BMEM memory area. The _first_ctbl is generated into the .ovly:_first_ctbl input section and the _second_ctbl is generated into the .ovly:_second_ctbl input section. Since the base names of these input sections match the name of the .ovly output section, the input sections are collected into the .ovly output section, which is then mapped to an address in the BMEM memory area.

If you do not provide explicit placement instructions for the linker-generated copy table sections, they are allocated according to the linker's default placement algorithm.

The linker does not allow other types of input sections to be combined with a copy table input section in the same output section. The linker does not allow a copy table section that was created from a partial link session to be used as input to a succeeding link session.

12.8.4.5 Splitting Object Components and Overlay Management

It is possible to split sections that have separate load and run placement instructions. The linker can access both the load address and run address of every piece of a split object component. Using the table() operator, you can tell the linker to generate this information into a copy table. The linker gives each piece of the split object component a COPY_RECORD entry in the copy table object.

For example, consider an application which has seven tasks. Tasks 1 through 3 are overlaid with tasks 4 through 7 (using a UNION directive). The load placement of all of the tasks is split among four different memory areas (LMEM1, LMEM2, LMEM3, and LMEM4). The overlay is defined as part of memory area PMEM. You must move each set of tasks into the overlay at run time before any services from the set are used.

You can use table() operators in combination with splitting operators, >>, to create copy tables that have all the information needed to move either group of tasks into the memory overlay as shown in the following example.

Creating a Copy Table to Access a Split Object Component

```
SECTIONS
{
   UNION
   {
      .task1to3: { *(.task1), *(.task2), *(.task3) }
                 load >> LMEM1 | LMEM2 | LMEM4, table(_task13_ctbl)
      GROUP
      {
         .task4: { *(.task4) }
         .task5: { *(.task5) }
         .task6: { *(.task6) }
         .task7: { *(.task7) }
      } load >> LMEM1 | LMEM3 | LMEM4, table(_task47_ctbl)
   } run = PMEM
   ...
   .ovly: > LMEM4
}
```

The following example illustrates a possible driver for such an application.

Copyright © 2026 Texas Instruments Incorporated

Split Object Component Driver

```
#include <cpy_tbl.h>
extern COPY_TABLE task13_ctbl;
extern COPY_TABLE task47_ctbl;
extern void task1(void);
...
extern void task7(void);
main()
{
   ...
   copy_in(&task13_ctbl);
   task1();
   task2();
   task3();
   ...
   copy_in(&task47_ctbl);
   task4();
   task5();
   task6();
   task7();
   ...
}
```

The contents of the .task1to3 section are split in the section's load space and contiguous in its run space. The linker-generated copy table, _task13_ctbl, contains a separate COPY_RECORD for each piece of the split section .task1to3. When the address of _task13_ctbl is passed to copy_in(), each piece of .task1to3 is copied from its load location into the run location.

The contents of the GROUP containing tasks 4 through 7 are also split in load space. The linker performs the GROUP split by applying the split operator to each member of the GROUP in order. The copy table for the GROUP then contains a COPY_RECORD entry for every piece of every member of the GROUP. These pieces are copied into the memory overlay when the _task47_ctbl is processed by copy_in().

The split operator can be applied to an output section, GROUP, or the load placement of a UNION or UNION member. The linker does not permit a split operator to be applied to the run placement of either a UNION or of a UNION member. The linker detects such violations, emits a warning, and ignores the offending split operator usage.

12.8.5 Compression

When automatically generating copy tables, the linker provides a way to compress the load-space data. This can reduce the read-only memory foot print. This compressed data can be decompressed while copying the data from load space to run space.

You can specify compression in two ways:

- The linker command line option --copy_compression=compression_kind can be used to apply the specified compression to any output section that has a table() operator applied to it.
- The table() operator accepts an optional compression parameter. The syntax is: .

  table(*name***, compression=** *compression_kind***)**

  The *compression_kind* can be one of the following types:
  - **off**. Do not compress the data.
  - **rle**. Compress data using Run Length Encoding.
  - **lzss**. Compress data using Lempel-Ziv-Storer-Szymanski compression.

A table() operator without the compression keyword uses the compression kind specified using the command line option --copy_compression. If no *compression_kind* was specified with the command-line option, the default is LZSS compression.

When you choose compression, it is not guaranteed that the linker will compress the load data. The linker compresses load data only when such compression reduces the overall size of the load space. In some cases

Copyright © 2026 Texas Instruments Incorporated

even if the compression results in smaller load section size the linker does not compress the data if the decompression routine offsets for the savings.

For example, assume RLE compression reduces the size of section1 by 30 bytes. Also assume the RLE decompression routine takes up 40 bytes in load space. By choosing to compress section1 the load space is increased by 10 bytes. Therefore, the linker will not compress section1. On the other hand, if there is another section (say section2) that can benefit by more than 10 bytes from applying the same compression then both sections can be compressed and the overall load space is reduced. In such cases the linker compresses both the sections.

You cannot force the linker to compress the data when doing so does not result in savings.

You cannot compress the decompression routines or any member of a GROUP containing .cinit.

12.8.5.1 Compressed Copy Table Format

The copy table format is the same irrespective of the *compression_kind*. The size field of the copy record is overloaded to support compression. Figure 12-4 illustrates the compressed copy table layout.

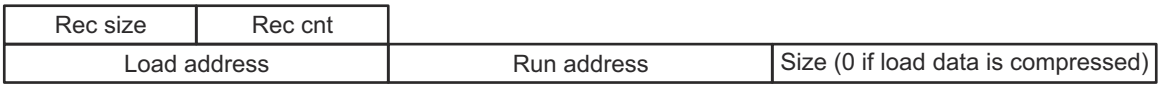


Figure 12-4. Compressed Copy Table

In Figure 12-4, if the size in the copy record is non-zero it represents the size of the data to be copied, and also means that the size of the load data is the same as the run data. When the size is 0, it means that the load data is compressed.

12.8.5.2 Compressed Section Representation in the Object File

The linker creates a separate input section to hold the compressed data. Consider the following table() operation in the linker command file.

```
SECTIONS
{
    .task1: load = ROM, run = RAM, table(_task1_table)
}
```

The output object file has one output section named .task1 which has different load and run addresses. This is possible because the load space and run space have identical data when the section is not compressed.

Alternatively, consider the following:

```
SECTIONS
{
    .task1: load = ROM, run = RAM, table(_task1_table, compression=rle)
}
```

If the linker compresses the .task1 section then the load space data and the run space data are different. The linker creates the following two sections:

- **.task1** : This section is uninitialized. This output section represents the run space image of section task1.
- **.task1.load** : This section is initialized. This output section represents the load space image of the section task1. This section usually is considerably smaller in size than .task1 output section.

The linker allocates load space for the .task1.load input section in the memory area that was specified for load placement for the .task1 section. There is only a single load section to represent the load placement of .task1 - .task1.load. If the .task1 data had not been compressed, there would be two allocations for the .task1 input section: one for its load placement and another for its run placement.

Copyright © 2026 Texas Instruments Incorporated

12.8.5.3 Compressed Data Layout

The compressed load data has the following layout:

-bit index	Compressed data

The first bits of the load data are the handler index. This handler index is used to index into a handler table to get the address of a handler function that knows how to decode the data that follows. The handler table is a list of 48-bit function pointers as shown in Figure 12-5.

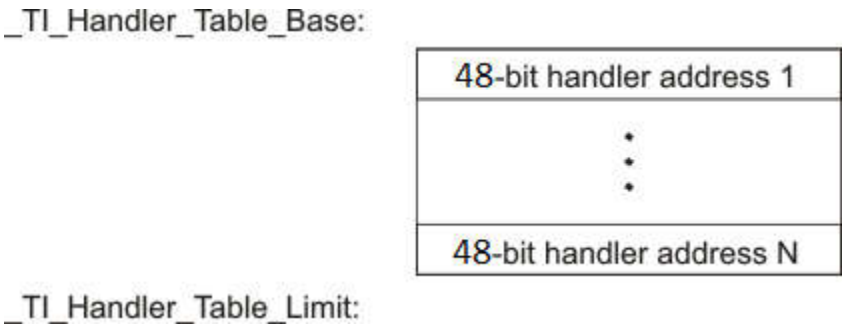


Figure 12-5. Handler Table

The linker creates a separate output section for the load and run space. For example, if .task1.load is compressed using RLE, the handler index points to an entry in the handler table that has the address of the run-time-support routine __TI_decompress_rle().

12.8.5.4 Run-Time Decompression

During run time you call the run-time-support routine copy_in() to copy the data from load space to run space. The address of the copy table is passed to this routine. First the routine reads the record count. Then it repeats the following steps for each record:

1. Read load address, run address and size from record.
2. If size is zero go to step 5.
3. Call memcpy passing the run address, load address and size.
4. Go to step 1 if there are more records to read.
5. Read the first from the load address.
6. Read the handler address from (&__TI_Handler_Base)[index].
7. Call the handler and pass load address + 1 and run address.
8. Go to step 1 if there are more records to read.

The routines to handle the decompression of load data are provided in the run-time-support library.

Copyright © 2026 Texas Instruments Incorporated

12.8.5.5 Compression Algorithms

The following subsections provide information about decompression algorithms for the RLE and LZSS formats. To see example decompression algorithms, refer to the following functions in the Run-Time Support library:

- **RLE:** The __TI_decompress_rle() function in the copy_decompress_rle.c file.
- **LZSS:** The __TI_decompress_lzss() function in the copy_decompress_lzss.c file.

Run Length Encoding (RLE):

-bit index	Initialization data compressed using run length encoding

The data following the -bit index is compressed using run length encoded (RLE) format. C7000 uses a simple run length encoding that can be decompressed using the following algorithm. See copy_decompress_rle.c for details.

1. Read the first , Delimiter (D).
2. Read the next (B).
3. If B != D, copy B to the output buffer and go to step 2.
4. Read the next (L).
   a. If L == 0, then length is either a value or we've reached the end of the data, read the next (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 -bit value (L).
5. Read the next (C); C is the repeat character.
6. Write C to the output buffer L times; go to step 2.
7. End of processing.

The C7000 run-time support library has a routine __TI_decompress_rle() to decompress data compressed using RLE. The first argument to this function is the address pointing to the after the -bit index. The second argument is the run address from the C auto initialization record.

Lempel-Ziv-Storer-Szymanski Compression (LZSS):

-bit index	Data compressed using LZSS

The data following the 8-bit index is compressed using LZSS compression. The C7000 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 after the -bit Index, and the second argument is the run address from the C auto initialization record.

See copy_decompress_lzss.c for details on the LZSS algorithm.

Copyright © 2026 Texas Instruments Incorporated

12.8.6 Copy Table Contents

To use a copy table generated by the linker, you must know the contents of the copy table. This information is included in a run-time-support library header file, cpy_tbl.h, which contains a C source representation of the copy table data structure generated by the linker.

C7000 cpy_tbl.h File

```
/*****************************************************************************/
/* cpy_tbl.h                                                                 */
/*                                                                           */
/* Copyright (c) 2003 Texas Instruments Incorporated     http://www.ti.com/  */
/*                                                                           */
/* Specification of copy table data structures which can be automatically    */
/* generated by the linker (using the table() operator in the LCF).          */
/*****************************************************************************/
#ifndef _CPY_TBL
#define _CPY_TBL
#include <stdint.h>
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
/*****************************************************************************/
/* Copy Record Data Structure                                                */
/*****************************************************************************/
typedef struct copy_record
{
   uint64_t load_addr;
   uint64_t run_addr;
   uint32_t size;
} __attribute((__packed__)) COPY_RECORD;
/*****************************************************************************/
/* Copy Table Data Structure                                                 */
/*****************************************************************************/
typedef struct copy_table
{
   uint16_t rec_size;
   uint16_t num_recs;
   COPY_RECORD    recs[1];
} COPY_TABLE;
/*****************************************************************************/
/* Prototype for general purpose copy routine.                               */
/*****************************************************************************/
extern void copy_in(COPY_TABLE *tp);
#ifdef __cplusplus
} /* extern "C" */
#endif /* __cplusplus */
#endif /* !_CPY_TBL */
```

For each object component that is marked for a copy, the linker creates a COPY_RECORD object for it. Each COPY_RECORD contains at least the following information for the object component:

- The load address
- The run address
- The size

The linker collects all COPY_RECORDs that are associated with the same copy table into a COPY_TABLE object. The COPY_TABLE object contains the size of a given COPY_RECORD, the number of COPY_RECORDs in the table, and the array of COPY_RECORDs in the table. For instance, in the BINIT example in Section 12.8.4.2, the .first and .extra output sections will each have their own COPY_RECORD entries in the BINIT copy table. The BINIT copy table will then look like this:

```
COPY_TABLE __binit__ = { 12, 2,
                         { <load address of .first>,
                           <run address of .first>,
                           <size of .first> },
                         { <load address of .extra>,
                           <run address of .extra>,
                           <size of .extra> } };
```


Copyright © 2026 Texas Instruments Incorporated

12.8.7 General Purpose Copy Routine

The cpy_tbl.h file in C7000 cpy_tbl.h File also contains a prototype for a general-purpose copy routine, copy_in(), which is provided as part of the run-time-support library. The copy_in() routine takes a single argument: the address of a linker-generated copy table. The routine then processes the copy table data object and performs the copy of each object component specified in the copy table.

The copy_in() function definition is provided in the cpy_tbl.c run-time-support source file shown in Run-Time-Support cpy_tbl.c File.

Run-Time-Support cpy_tbl.c File

```
/*****************************************************************************/
/* cpy_tbl.c                                                                 */
/*                                                                           */
/* Copyright (c) 2011 Texas Instruments Incorporated                         */
/*                                                                           */
/* General purpose copy routine. Given the address of a link-generated       */
/* COPY_TABLE data structure, effect the copy of all object components       */
/* that are designated for copy via the corresponding LCF table() operator.  */
/*                                                                           */
/*****************************************************************************/
#include <cpy_tbl.h>
#include <string.h>
typedef void (*handler_fptr)(const unsigned char *in, unsigned char *out);
/*****************************************************************************/
/* COPY_IN()                                                                 */
/*****************************************************************************/
void copy_in(COPY_TABLE *tp)
{
   unsigned short I;
   for (I = 0; I < tp->num_recs; I++)
   {
      COPY_RECORD crp = tp->recs[i];
      unsigned char *ld_addr = (unsigned char *)crp.load_addr;
      unsigned char *rn_addr = (unsigned char *)crp.run_addr;
      if (crp.size)
      {
         /*---------------------------------------------------------------*/
         /* Copy record has a non-zero size so the data is not compressed.*/
         /* Just copy the data.                                           */
         /*---------------------------------------------------------------*/
         memcpy(rn_addr, ld_addr, crp.size);
      }
#ifdef __TI_EABI__
      else if (HANDLER_TABLE)
      {
         /*---------------------------------------------------------------*/
         /* Copy record has size zero so the data is compressed. The first*/
         /* byte of the load data has the handler index. Use this index with */
         /* the handler table to get the handler for this data. Then call */
         /* the handler by passing the load and run address.              */
         /*---------------------------------------------------------------*/
         unsigned char index = *((unsigned char *)ld_addr++);
         handler_fptr hndl = (handler_fptr)(&HANDLER_TABLE)[index];
         (*hndl)((const unsigned char *)ld_addr, (unsigned char *)rn_addr);
      }
#endif
   }
}
```


Copyright © 2026 Texas Instruments Incorporated

12.9 Partial (Incremental) Linking

An output file that has been linked can be linked again with additional modules. This is known as *partial linking* or *incremental linking*. Partial linking allows you to partition large applications, link each part separately, and then link all the parts together to create the final executable program. Follow these guidelines for producing a file that you will relink:

- The intermediate files produced by the linker *must* have relocation information. Use the --relocatable option when you link the file the first time. (See Section 12.4.3.2.)
- Intermediate files *must* have symbolic information. By default, the linker retains symbolic information in its output. Do not use the --no_sym_table option if you plan to relink a file, because --no_sym_table strips symbolic information from the output module. (See Section 12.4.21.)
- Intermediate link operations should be concerned only with the formation of output sections and not with allocation. All allocation, binding, and MEMORY directives should be performed in the final link. Since the ELF object file format is used, input sections are not combined into output sections during a partial link unless a matching SECTIONS directive is specified in the link step command file.
- If the intermediate files have global symbols that have the same name as global symbols in other files and you want them to be treated as static (visible only within the intermediate file), you must link the files with the --make_static option (see Section 12.4.16.1).
- If you are linking C code, do not use --ram_model or --rom_model until the final linker. Every time you invoke the linker with the --ram_model or --rom_model option, the linker attempts to create an entry point. (See Section 12.4.24, Section 9.3.2.1, and Section 9.3.2.2.)

The following example shows how you can use partial linking:

Step 1: Link the file file1.com; use the --relocatable option to retain relocation information in the output file out1.out.

```
cl7x --run_linker --relocatable --output_file=out1 file1.com
```

file1.com contains:

```
SECTIONS
{
   ss1:   {
          f1.c.obj
          f2.c.obj
           ...
          fn.c.obj
          }
}
```

Step 2: Link the file file2.com; use the --relocatable option to retain relocation information in the output file out2.out. file2.com contains:

```
SECTIONS
{
   ss2:   {
          g1.c.obj
          g2.c.obj
           ...
          gn.c.obj
          }
}
```

Step 3: Link out1.out and out2.out.

```
cl7x --run_linker --map_file=final.map --output_file=final.out out1.out out2.out
```

12.10 Linking C/C++ Code

The C/C++ compiler produces assembly language source code that can be assembled and linked. For example, a C program consisting of modules prog1, prog2, etc., can be assembled and then linked to produce an executable file called prog.out:

```
cl7x --run_linker --rom_model --output_file prog.out prog1.c.obj prog2.c.obj ... rts7100_le.lib
```

The --rom_model option tells the linker to use special conventions that are defined by the C/C++ environment.

Copyright © 2026 Texas Instruments Incorporated

The archive libraries shipped by TI contain C/C++ run-time-support functions.

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.

12.10.1 Run-Time Initialization

C/C++ programs must be linked with code to initialize and execute the program, called a *bootstrap* routine (the *boot.c.obj* object module). The _c_int00 symbol is defined as the program entry point and is the start of the C boot routine in boot.c.obj. Referencing _c_int00 ensures that boot.c.obj is automatically linked in from the run-time-support library. When a program runs, it first executes boot.c.obj. The boot.c.obj symbol contains code and data to initialize the run-time environment; it performs the following:

- Set up the system stack and configuration registers
- Process run-time *.cinit* initialization table and autoinitialize global variables (if --rom_model was used)
- Disable interrupts and call _main

The run-time-support object libraries contain boot.c.obj. You can:

- Use the archiver to extract boot.c.obj from the library and then link the module in directly.
- Include the appropriate run-time-support library as an input file (the linker automatically extracts boot.c.obj when you use the --ram_model or --rom_model option).

12.10.2 Object Libraries and Run-Time Support

Chapter 7 describes additional run-time-support functions that are included in rts.src. If your program uses any of these functions, you must link the appropriate run-time-support library with your object files.

You can also create your own object libraries and link them. The linker includes and links only those library members that resolve undefined references.

12.10.3 Setting the Size of the Stack and Heap Sections

The C/C++ language uses two uninitialized sections called .sysmem and .stack for the memory pool used by the malloc( ) functions and the run-time stacks, respectively. You can set the size of these by using the --heap_size or --stack_size option and specifying the size of the section as a constant immediately after the option. If the options are not used, the default size of the heap is 1K bytes and the default size of the stack is 1K bytes.

See Section 12.4.13 for setting heap sizes and Section 12.4.27 for setting stack sizes.

12.10.4 Initializing and Autoinitializing 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. See Section 9.3.2.1 for details.

Initialization of variables at load time enhances performance by reducing boot time and saving memory used by initialization tables. To use this method, invoke the linker with --ram_model. See Section 9.3.2.2 for details.

See Section 9.3.2.3 for steps performed when you invoke the linker with --ram_model or --rom_model.

12.10.5 Constraints Due to CMMU Configuration

The C7000 compiler may generate vector predicated stores during vectorization that extend beyond the end of an address by up to the vector width minus one. For example, for C7100 devices, which have 64-byte vectors, a vector predicated store may extend an additional 63 bytes past the last valid address. For a list of C7000 vector widths, see Table 5-7.

If the bytes after a range defined by the MEMORY directive may be inaccessible at run-time due to the Corepac Memory Management Unit (CMMU) configuration, the length of that memory range should be reduced to prevent placement of a section in memory that may cause a page fault at run-time due to a vector predicated store. For example, for C7100 devices, reduce the length of the potentially affected memory range by 63 bytes.

For more information about the MEMORY directive, see Section 12.5.4. For more information about vectorization, see the *C7000 C/C++ Optimization Guide* (SPRUIV4).

Copyright © 2026 Texas Instruments Incorporated

12.11 Linker Example

This example links three object files named demo.c.obj, filter.c.obj, and tables.obj and creates a program called demo.out.

Assume that target memory has the following program memory configuration:

Address Range	Contents
0x0080 to 0x7000	On-chip RAM_PG
0xC000 to 0xFF80	On-chip ROM

Address Range	Contents
0x0080 to 0x0FFF	RAM block ONCHIP
0x0060 to 0xFFFF	Mapped external addresses EXT

Address Range	Contents
0x00000020 to 0x00210000	PMEM
0x00400000 to 0x01400000	EXT0
0x01400000 to 0x01800000	EXT1
0x02000000 to 0x03000000	EXT2
0x03000000 to 0x04000000	EXT3
0x40000000 to 0x82000000	BMEM

The output sections are constructed in the following manner:

- Executable code, contained in the .text sections of demo.c.obj, filters.c.obj, as well as executable code from the RTS library, are linked into program memory PMEM.
- Two data objects are defined in tables.c.obj. Each is placed in its own output section: .tableA and .tableB. When the program is loaded, both the .tableA and .tableB output sections are linked into separate locations in the BMEM area. However, run-time access to these tables refers to the run-time location indicated by the symbol "filter_matrix", which is defined as the start address of the UNION containing both .tableA and .tableB. This location is linked into the EXT1 memory area. At run-time, the application is responsible for copying either .tableA or .tableB from its load location in BMEM to its run location in EXT1 before attempting to access data from the table that was copied. The linker supports the copy table mechanisms described in Section 12.8 to help facilitate this action.
- For architectures that support and use DP-relative addressing, all data objects that are accessed using DP-relative addressing are collected into a group consisting of the .bss output sections. This group is linked into the BMEM memory area.
- Since the demo.out program uses command line arguments that must be specified when demo.out is loaded and run, the application must reserve space for passing command-line arguments to the program in the .args section. The amount of space allocated for the .args section is indicated in the '--args 0x1000' option near the top of the linker command file. The .args output section is then linked into the BMEM memory area. Support for processing command-line arguments is provided in the boot routine contained in the RTS library that will be linked into the demo.out program.
- The size of the software stack is indicated by the "--stack 0x6000" option near the top of the linker command file. Likewise, the size of the heap, from which memory can be dynamically allocated at run-time, is indicated via the "--heap 0x3000" option near the top of the linker command file. Both the .stack and .sysmem (which contains the heap) output sections are linked into the BMEM memory area.

Linker Command File, mylnk.cmd shows the linker command file for this example. Output Map File, demo.map shows the map file.

Copyright © 2026 Texas Instruments Incorporated

Linker Command File, mylnk.cmd

```
/****************************************************************************/
/*** Specify Linker Options                                               ***/
/****************************************************************************/
-cr                                 /* --ram_model: load-time initialization */
--heap 0x3000
--stack 0x6000
--args 0x1000
--output_file=demo.out                        /* Name the output file      */
--map_file=demo.map                           /* Create an output map file */
--undefined_sym=filter_table_A              /* Introduce an undefined symbol */
--undefined_sym=filter_table_B              /* Introduce an undefined symbol */
/****************************************************************************/
/*** Specify the Input Files                                              ***/
/****************************************************************************/
demo.c.obj
tables.obj
filter.c.obj
/****************************************************************************/
/*** Specify Runtime Support Library to be linked in                      ***/
/****************************************************************************/
-l libc.a
/****************************************************************************/
/*** Specify the Memory Configuration                                     ***/
/****************************************************************************/
MEMORY
{
  PMEM: o = 00000020h l = 0020ffe0h
  EXT0: o = 00400000h l = 01000000h
  EXT1: o = 01400000h l = 00400000h
  EXT2: o = 02000000h l = 01000000h
  EXT3: o = 03000000h l = 01000000h
  BMEM: o = 40000000h l = 02000000h
}
/****************************************************************************/
/*** Specify the Output Sections                                          ***/
/****************************************************************************/
SECTIONS
{
  .text : > PMEM
  UNION
  {
    .tableA: { tables.obj(tableA) } load > BMEM, table(tableA_cpy)
    .tableB: { tables.obj(tableB) } load > BMEM, table(tableB_cpy)
  } RUN = EXT1, RUN_START(filter_matrix)
  GROUP
  {
    .rodata:
    .bss:
  } > EXT2
  .stack:  > BMEM
  .args :  > BMEM
  .cinit:  > BMEM
  .cio:    > BMEM
  .const:  > BMEM
  .data:   > BMEM
  .sysmem: > BMEM
}
/****************************************************************************/
/*** End of Command File                                                  ***/
/****************************************************************************/
```

Invoke the linker by entering the following command:

```
cl7x --run_linker mylnk.cmd
```

This creates the map file shown in Output Map File, demo.map and an output file called demo.out that can be run on a C7000.

Copyright © 2026 Texas Instruments Incorporated

Output Map File, demo.map

```
OUTPUT FILE NAME:   <demo.out>
ENTRY POINT SYMBOL: "_c_int00"  address: 000000007ec0
MEMORY CONFIGURATION
         name             origin         length     used     unused   attr    fill
----------------------  ------------   --------  --------  --------  ----  --------
  PMEM                  000000000020   0020ffe0  00008170  00207e70  RWIX
  EXT0                  000000400000   01000000  00000000  01000000  RWIX
  EXT1                  000001400000   00400000  00000080  003fff80  RWIX
  EXT2                  000002000000   01000000  000000b0  00ffff50  RWIX
  EXT3                  000003000000   01000000  00000000  01000000  RWIX
  BMEM                  000040000000   02000000  0000a6e4  01ff591c  RWIX
SEGMENT ALLOCATION MAP
  run origin     load origin      length   init length attrs members
--------------  --------------  ---------- ----------- ----- -------
000000000040    000000000040    00008170   00008170    r-x
  000000000040    000000000040    00008140   00008140    r-x .text
  000000008180    000000008180    00000030   00000030    r-- .ovly
000001400000    00004000a5e4    00000080   00000080    rw-
  000001400000    00004000a5e4    00000080   00000080    rw- .tableA
000001400000    00004000a664    00000080   00000080    rw-
  000001400000    00004000a664    00000080   00000080    rw- .tableB
000002000000    000002000000    000000b0   00000000    rw-
  000002000000    000002000000    000000b0   00000000    rw- .bss
000040000000    000040000000    00009000   00000000    rw-
  000040000000    000040000000    00006000   00000000    rw- .stack
  000040006000    000040006000    00003000   00000000    rw- .sysmem
000040009000    000040009000    00001384   00001384    rw-
  000040009000    000040009000    00001000   00001000    rw- .args
  00004000a000    00004000a000    00000384   00000384    rw- .data
00004000a384    00004000a384    00000140   00000140    r--
  00004000a384    00004000a384    00000140   00000140    r-- .const
00004000a4c4    00004000a4c4    00000120   00000000    rw-
  00004000a4c4    00004000a4c4    00000120   00000000    rw- .cio
SECTION ALLOCATION MAP
 output                                    attributes/
section   page      origin       length       input sections
--------  ----  -------------  ----------   ----------------
.text       0    000000000040   00008140
                   000000000040   00002400     rts7100_le.lib : _printfi.c.obj (.text:__TI_printfi)
                   000000002440   00000900                    : _printfi.c.obj (.text:_pconv_a)
                   000000002d40   00000800                    : _printfi.c.obj (.text:_pconv_g)
                   000000003540   00000680                    : _printfi.c.obj (.text:_pconv_e)
                   000000003bc0   000003c0                    : memory.c.obj (.text:aligned_alloc)
                   000000003f80   00000380                    : frcdivd.c.obj (.text:__TI_frcdivd)
                   000000004300   00000340                    : _printfi.c.obj (.text:_pconv_f)
                   000000004640   00000300                    : _printfi.c.obj (.text:fcvt)
                   000000004940   00000300                    : fputs.c.obj (.text:fputs)
...
.rodata     0    000002000000   00000000     UNINITIALIZED
.bss        0    000002000000   000000b0     UNINITIALIZED
                   000002000000   000000a0     (.common:__TI_tmpnams)
                   0000020000a0   00000008     rts7100_le.lib : memory.c.obj (.bss)
                   0000020000a8   00000008     (.common:parmbuf)
.stack      0    000040000000   00006000     UNINITIALIZED
                   000040000000   00000010     rts7100_le.lib : boot.c.obj (.stack)
                   000040000010   00005ff0     --HOLE--
.sysmem     0    000040006000   00003000     UNINITIALIZED
                   000040006000   00000010     rts7100_le.lib : memory.c.obj (.sysmem)
                   000040006010   00002ff0     --HOLE--
.args       0    000040009000   00001000
                   000040009000   00001000     --HOLE-- [fill = 0]
.data       0    00004000a000   00000384
                   00004000a000   000001e0     rts7100_le.lib : defs.c.obj (.data:_ftable)
                   00004000a1e0   000000d8                    : host_device.c.obj (.data:_device)
                   00004000a2b8   000000a0                    : host_device.c.obj (.data:_stream)
                   00004000a358   00000010                    : exit.c.obj (.data)
                   00004000a368   00000008                    : _lock.c.obj (.data:_lock)
                   00004000a370   00000008                    : _lock.c.obj (.data:_unlock)
                   00004000a378   00000004                    : defs.c.obj (.data)
                   00004000a37c   00000004                    : errno.c.obj (.data)
                   00004000a380   00000004                    : memory.c.obj (.data)
.const      0    00004000a384   00000140
                   00004000a384   00000101     rts7100_le.lib : ctype.c.obj
(.const:.string:_ctypes_)
```


Copyright © 2026 Texas Instruments Incorporated

```
                  00004000a485    00000003     --HOLE-- [fill = 0]
                  00004000a488    00000024                  : _printfi.c.obj (.const:.string)
                  00004000a4ac    00000018     demo.c.obj (.const:.string)
.tableA    0    00004000a5e4    00000080     RUN ADDR = 000001400000
                  00004000a5e4    00000080     tables.obj (tableA)
.tableB    0    00004000a664    00000080     RUN ADDR = 000001400000
                  00004000a664    00000080     tables.obj (tableB)
.cinit     0    000040000000    00000000     UNINITIALIZED
.cio       0    00004000a4c4    00000120     UNINITIALIZED
                  00004000a4c4    00000120     rts7100_le.lib : trgmsg.c.obj (.cio)
.ovly      0    000000008180    00000030
                  000000008180    00000018     (.ovly:tableA_cpy)
                  000000008198    00000018     (.ovly:tableB_cpy)
...
LINKER GENERATED COPY TABLES
tableA_cpy @ 00008180 records: 1, size/record: 20, table size: 24
        .tableA: load addr=00004000a5e4, load size=00000080, run addr=000001400000,
         run size=00000080, compression=none
tableB_cpy @ 00008198 records: 1, size/record: 20, table size: 24
        .tableB: load addr=00004000a664, load size=00000080, run addr=000001400000,
         run size=00000080, compression=none
GLOBAL SYMBOLS: SORTED ALPHABETICALLY BY Name
address        name
-------------  ----
000000008040   C$$EXIT
000000006eb0   C$$IO$$
0000000069c0   HOSTclose
000000006100   HOSTlseek
000000005f80   HOSTopen
...
000000008180   tableA_cpy
000000008198   tableB_cpy
000000007cc0   unlink
000000007d40   wcslen
000000007dc0   write
...
[96 symbols]
```

12.12 Static Shared Objects in Multicore Applications

The C7000 compiler tools support OptiShare, a multicore memory-sharing feature. OptiShare uses Static Shared Objects (SSO) to allow multiple C7000 cores to share identical code and data sections. These sections can be shared by multiple applications running on TI core processors in a multicore system.

SSOs help reduce the overall size of the code and data loaded into local and shared memory on TI devices that have the following characteristics:

- **Multiple Identical TI Core Processors.** A TI device with two or more identical core processors that can execute code and access data from local core or shared system memory.
- **Ability to Execute Code and Access Shared Read-Only (RO) Data from Shared System Memory.** Each core processor on the TI device must be able to fetch instructions and access RO data directly from shared system memory.
- **Shared System to Local Core Memory Mapping for Shared Read-Write (RW) Data Objects.** Each of the identical TI core processors can map an address range from shared system memory to local memory that is accessible only to an individual TI core processor. For example, if a multicore system places a global read-write (RW) data object in shared system memory, then each application must maintain its own copy of the global RW data object in local core memory that is referenced using its shared memory address. An MMU or similar mechanism can be used to perform such a mapping.

Copyright © 2026 Texas Instruments Incorporated

Note

Limitations of SSO: The following limitations apply to OptiShare:

- **Scalar Data Not Shareable:** Scalar global variables are not placed in their own subsections. All the scalars in one object file share a single .data section. OptiShare shares that section only when its content is identical on every core.
- **Link Time Optimization Not Available with SSO:** Link Time Optimization and Shared Static Objects are mutually incompatible. A warning is provided if you attempt to enable LTO when building an SSO file or linking with an SSO file.
- **Load and Initialize SSO Content Before Execution:** The contents of the SSO file must be loaded and initialized in system memory prior to executing any code that can access any function or data object defined in the SSO.

For details about local core and shared system memory available on C7000 devices, see target-specific documentation.

At a high level, the development flow for this feature is as follows:

1. Compile and link initial individual applications, generating an XML link information file for each.
2. Identify functions, RO data objects, and RW data objects that are common among the individual applications.
3. Collect common functions, RO data objects, and RW data objects into a Shared Static Object (SSO) to be loaded into shared system memory.
4. Re-link each individual application against the SSO from step 3, allocating code and data that is NOT defined in the SSO to local core memory.

The following sections explore the creation of an example SSO following this development flow.

12.12.1 SSO Example

For this example, assume there are two TI C7100 processors, each running a simple application. The source for the first application is contained in sub-directory `core1`:

```
/* main.c */
    extern void do_some_work(void);

    int main() {
      do_some_work();
      return 0;
    }

    /* do_some_work.c */
    volatile int xyz[4] = { 10, 20, 30, 40 };

    void do_some_work(void) {
      xyz[0] += 2;
    }
```

Notice that `xyz` is an array, not a scalar. The compiler puts an array in its own subsection (`.data:xyz`), so OptiShare can share it alone. Scalar variables stay in the combined `.data` section.

For the sake of simplicity, assume that the `core2` sub-directory contains identical copies of the above source files.

12.12.2 Step 1: Compile and Link Applications

To identify functions and data objects that are common to multiple applications in the multicore system, first gather information about these objects to determine whether they are equivalent. To this end, the linker provides the `--gen_xml_func_hash` option. When used in combination with the `--xml_link_info=` linker option, the linker computes a hash value for each function and data object defined in an application. Each section in the generated XML link information file that contains a function or data object definition is annotated with its hash value.

Copyright © 2026 Texas Instruments Incorporated

For this example, compile and link the core1 application with the following commands:

```
%> cl7x -mv7100 --gen_func_subsections main.c
%> cl7x -mv7100 --gen_func_subsections do_some_work.c
%> cl7x -mv7100 --gen_func_subsections -z main.obj do_some_work.obj linker.cmd \
      --xml_link_info=app1.xml --gen_xml_func_hash --ram_model -o app1.out --map_file=app1.map
```

Likewise, compile and link the core2 application with the following commands:

```
%> cl7x -mv7100 --gen_func_subsections main.c
%> cl7x -mv7100 --gen_func_subsections do_some_work.c
%> cl7x -mv7100 --gen_func_subsections -z main.obj do_some_work.obj linker.cmd \
      --xml_link_info=app2.xml --gen_xml_func_hash --ram_model -o app2.out --map_file=app2.map
```

The output files from these two links that are used in the next step of the development flow are `app1.xml` and `app2.xml`.

Notice the following compiler options:

- The `--gen_func_subsections` compiler option places each function in a separate subsection in the object file. Without this option, all functions in one object file are placed in a single `.text` section, and OptiShare can share only the entire section if it is identical across cores.
- The `--gen_data_subsections=on` compiler option is on by default, so it does not appear in the example commands above. It places each struct and each array in its own subsection of the `.data` section. Scalar variables remain in the combined .data section. If you use `--gen_data_subsections=off`, all data objects are placed in a single `.data` section, and OptiShare can share the entire section if it is identical across cores.

To exclude a specific function or variable from OptiShare, use the `do_not_share` attribute. Symbols marked with this attribute are never placed in the SSO even if their content is identical across all input executables. For example:

```
__attribute__((do_not_share)) void sensitive_func(void) { ... }

__attribute__((do_not_share)) int my_var;
```

If you mark a scalar variable with `__attribute__((do_not_share))`, OptiShare does not share the whole `.data` section of that object file. It also does not share any function that uses data in that section.

12.12.3 Step 2: Identify Common Objects

The next step in the development flow is to identify the functions and data objects that are common to two or more of the applications that participate in the multicore system.

The C7000 compiler tools installation includes an `opti-share` sub-directory for this task. This sub-directory contains the `opti-share.js` JavaScript file, which can process the XML link information files generated by the initial application builds. This script identifies common functions and data objects that can be placed in shared system memory. The `opti-share.js` script uses an additional `--mem_spec=` input option to determine the placement of the shared functions, read-only (RO) data objects, and read-write (RW) data objects. As output, it generates an SSO linker command file.

Before running the `opti-share.js` script to create the SSO linker command file for this example, it is useful to examine the contents of the XML link information files, `app1.xml` and `app2.xml`, to understand the information used by the `opti-share.js` script to identify functions and data objects that are common to both `app1.out` and `app2.out`.

Copyright © 2026 Texas Instruments Incorporated

In app1.xml, there are object_component XML tags that represent the sections where main(), do_some_work(), and xyz are defined:

```
%> cat app1.xml
    <?xml version="1.0" encoding="ISO-8859-1" ?>
    <link_info>
       <banner>C7x Linker Unix v6.2.0.LTS</banner>
       ...
       <object_component_list>
          ...
             <object_component id="oc-9d">
             <name>.text:main</name>
             <load_address>0x700702c0</load_address>
             <readonly>true</readonly>
             <executable>true</executable>
             <run_address>0x700702c0</run_address>
             <size>0x40</size>
             <alignment>0x40</alignment>
             <input_file_ref idref="fl-1"/>
             <refd_ro_sections>
                <object_component_ref idref="oc-d3"/>
             </refd_ro_sections>
             <value>f273b1cfcadee5428fb34264b24c6a56</value>
          </object_component>
          ...
          <object_component id="oc-d3">
             <name>.text:do_some_work</name>
             <load_address>0x70070240</load_address>
             <readonly>true</readonly>
             <executable>true</executable>
             <run_address>0x70070240</run_address>
             <size>0x40</size>
             <alignment>0x40</alignment>
             <input_file_ref idref="fl-2"/>
             <refd_rw_sections>
                <object_component_ref idref="oc-ee"/>
             </refd_rw_sections>
             <value>2be4772f025fff731d8e0fe9e455422a</value>
          </object_component>
          ...
          <object_component id="oc-ee">
             <name>.data:xyz</name>
             <load_address>0x40001010</load_address>
             <readwrite>true</readwrite>
             <run_address>0x40001010</run_address>
             <size>0x10</size>
             <alignment>0x4</alignment>
             <input_file_ref idref="fl-2"/>
             <value>1e518b7adf404f97391b71f8992eb256</value>
          </object_component>
          ...
       </object_component_list>
       ...
       <title>Link successful</title>
    </link_info>
```

Likewise, in app2.xml, the sections containing the definitions of main(), do_some_work(), and xyz are also represented by object_component XML tags. In a real-world multicore system, the addresses for <load_address> and <run_address> would differ, but the hash values would remain the same. Matching hash values indicate to OptiShare that the objects are equivalent.

A few things to observe about the contents of app1.xml and app2.xml:

- The hash value computed for a function or data object is found in the value member of the object_component, which represents the section where the function or data object is defined:
  - main() is defined in section .text:main
  - do_some_work() is defined in section .text:do_some_work
  - xyz is defined in section .data:xyz
- The hash values for main(), do_some_work(), and xyz in app1.xml match the hash values for main(), do_some_work(), and xyz in app2.xml even though the addresses where they are loaded differ

Copyright © 2026 Texas Instruments Incorporated

between app1.xml and app2.xml. Matching hash values indicate that the implementations of main(), do_some_work(), and xyz are identical between app1 and app2.

- The refd_ro_sections and refd_rw_sections lists that appear in <object_component> records facilitate a reconstruction of the application's section reference graph. In this example specifically:
  - The <object_component> records for main() in both app1.xml and app2.xml contain a refd_ro_sections list that enumerates the sections (or <object_component> records) that are referenced from main(), namely the .text:do_some_work section, which represents the call from main() to do_some_work().
  - The <object_component> records for do_some_work() in both app1.xml and app2.xml contain a refd_rw_sections list that enumerates the sections (or object_component records) that are referenced from do_some_work(), namely the .data:xyz section, which represents the read and write of xyz from do_some_work().

12.12.4 Step 3: Create a Static Shared Object (SSO) File

Invoke the opti-share.js script using Node.js. For this example, the command line is as follows:

```
%> node /path/to/install/opti-share/opti-share.js -o sso.cmd core1/app1.xml core2/app2.xml \
      --mem_spec=ex_mem_spec_c7x.json
```

The arguments to the opti-share.js invocation are as follows:

- **-o sso.cmd** - specifies the name of the linker command file to be written by the opti-share.js script
- **core1/app1.xml core2/app2.xml** - the path to and name of the XML link information files considered by the opti-share.js script to identify common functions and data objects
- **--mem_spec=ex_mem_spec.json** - JSON file containing a specification of available shared memory areas and placement instructions for the collections of common functions, RO data objects, initialized RW data objects, and uninitialized RW data objects. This file is provided as part of the C7000 compiler tools installation.

This command results in an SSO linker command file, sso.cmd, with the following content:

```
%> cat sso.cmd

--map_file=sso.map
--output_file=sso.out
--no_entry_point
--sso

MEMORY {
  SHARED_RX: o=0x70070000 l=0x00006000
  SHARED_RO: o=0x70076000 l=0x00004000
  SHARED_RW: o=0x7007A000 l=0x00006000
}

SECTIONS {
  .shared.text {
    . = align(64); "/path/to/install/lib/rts7100_le.lib"<exit.c.obj>(.text:abort)
    . = align(64); "/path/to/install/lib/rts7100_le.lib"<exit.c.obj>(.text:exit)
    . = align(64); "/path/to/install/lib/rts7100_le.lib"<pre_init.c.obj>(.text:_system_pre_init)
    . = align(64); "/path/to/install/lib/rts7100_le.lib"<startup.c.obj>(.text:_system_post_cinit)
    . = align(64); "/path/to/example/core1/do_some_work.obj"(.text:do_some_work)
    . = align(64); "/path/to/example/core1/main.obj"(.text:main)
  } > SHARED_RX, priority(4)

  .shared.data {
    . = align(4); "/path/to/example/core1/do_some_work.obj"(.data:xyz)
    . = align(8); "/path/to/install/lib/rts7100_le.lib"<exit.c.obj>(.data)
  } > SHARED_RW, priority(1)

  .shared.bss {
    . = align(8); "/path/to/install/lib/rts7100_le.lib"<boot.c.obj>(.stack)
  } > SHARED_RW, priority(2)
}
```


Copyright © 2026 Texas Instruments Incorporated

12.12.4.1 SSO Linker Command File Content from ex_mem_spec.json

As mentioned above, the MEMORY directive is dictated by the `ex_mem_spec.json` file, which describes the available shared memory regions. This file is installed in the `opti-share` sub-directory. The `ex_mem_spec.json` file also contains placement instructions for the four potential output sections that the `opti-share.js` script can write into the SECTIONS directive of the generated SSO linker command file. These include:

- **.shared.text** - the collection of common functions placed in shared memory
- **.shared.const** - the collection of common RO data objects placed in shared memory
- **.shared.data** - the collection of initialized RW data objects placed in shared memory
- **.shared.bss** - the collection of uninitialized RW data objects placed in shared memory

Note

Shared RW Data Objects Get Duplicated in Local Core Memory: If `opti-share.js` determines that a RW data object is eligible to go into shared memory, a subsequent link of the application against the SSO file allocates a local copy of the RW data object, since each application must maintain its own copy of the shared RW data object. Such local copies can reduce code size savings realized via shared functions. However, the advantage of sharing a RW data object is that it can enable a greater number of common functions to be placed in shared memory. To avoid sharing of RW data objects, there are two additional `opti-share.js` options:

- **--nodata** - when specified, `opti-share.js` does not share any initialized RW data objects
- **--nobss** - when specified, `opti-share.js` does not share any uninitialized RW data objects

12.12.4.2 SSO Linker Command File Content from opti-share.js

The `opti-share.js` script is responsible for identifying common functions and data objects to be placed in shared memory. This script uses the following rules to determine which functions and data objects are eligible for placement in shared memory:

- In order for two functions or two data objects to be considered identical, their hash values must match exactly.
- There must be two or more identical instances of a function or data object among the individual applications participating in the multicore system.
- Any function or data object referenced by a candidate common function must be ruled a common function or data object.

To emphasize the significance of the third bullet above, be aware of the following note.

Note

Static Shared Object Defined: The essential characteristic of a *static shared object* is that all function or data object symbol references from functions that are defined in a static shared object must also be defined in the same static shared object. For example, in the example under consideration, if `do_some_work()` were implemented differently in `app1.out` vs. `app2.out`, then `main()` would be ruled ineligible to be placed in shared memory since a function that it references is not identical in both `app1.out` and `app2.out`.

Once all eligible common functions and data objects have been identified, *opti-share.js* populates the content of the output section specifications in the SSO linker command file with the full path name for each input section that contains the definition of a common function or data object.

Note

Development Flow Assumes the File System Remains Consistent: The full path name is referenced in the SSO linker command file. Therefore, the file system organization must remain consistent from the initial compilation to the link of the individual applications to the relink of each application against the SSO.

Copyright © 2026 Texas Instruments Incorporated

12.12.4.3 Build the SSO File

To build an SSO file, invoke the linker with the SSO linker command file as the only argument:

```
%> cl7x -z sso.cmd --xml_link_info=sso.lnkxml
```

The `--xml_link_info` linker option causes the linker to compute a hash value for each function and data object defined in an application. This is useful for running the `opti-share.js` script against the SSO itself.

In the above SSO linker command file content, the following linker options are already specified:

```
--map_file=sso.map
--output_file=sso.out
--no_entry_point
--sso
```

That is, the linker is already instructed to generate a map file called `sso.map` and an SSO output file called `sso.out`. The `--no_entry_point` option instructs the linker to create an ELF executable object file that does not have a defined entry symbol. Finally, the `--sso` option informs the linker that it is in the process of generating an SSO output file.

12.12.5 Step 4: Relink Applications Against the SSO

Finally, it is time to relink each individual application against the SSO. This step in the development flow allows each individual application to use the existing shared version of a common function or RO data object without having to include its own version of the same function or RO data object in the linked output file.

The C7000 command to perform the relink is similar to the command used in the initial link of the application:

```
%> cl7x -z --import_sso=../sso.out --ram_model main.obj do_some_work.obj linker.cmd \
      -o app1_sso.out --map_file=app1_sso.map
```

The differences are the addition of the `--import_sso=sso.out` option and the removal of the `--xml_link_info` and `--gen_xml_func_hash` options.

Adding the `--import_sso=sso.out` option has these effects on the link:

- The linker imports all globally exported symbols in the SSO, not only shared symbols. If a symbol that should not be shared is in the SSO by mistake (for example, because of a prefix match on a .data rule or an omitted do_not_share attribute), each core link imports the symbol and overwrites the core-local definition. You can correct this issue by adding the `--localize=`*`pattern`* linker option to the SSO link, which removes the global export so that `--import_sso` cannot resolve it.
- During the symbol resolution phase of the link, symbols defined in the SSO file are preferred to definitions of the same symbols from elsewhere.
- If the SSO file contains definitions of RW data objects, the linker allocates space in a local core region of memory for a duplicate copy of each shared RW data object.
- If the SSO file contains definitions of RW data objects, the linker emits a set of absolute symbols that can be referenced by code that is responsible for programming the MMU to establish address mappings between:
  - the .shared.data output section location and size in the SSO and the application's copy of the .shared.data section
  - the .shared.bss output section location and size in the SSO and the application's copy of the .shared.bss section
- RW data sharing requires MMU address translation, which must be configured by SDK boot code for each core.

The `--xml_link_info` and `--gen_xml_func_hash` options are removed because the relink is not expected to feed the creation of another OptiShare SSO. These options are not harmful and could be included.

Copyright © 2026 Texas Instruments Incorporated

12.12.5.1 Application .out and SSO File Content

You can examine the disassembly output from `app1_sso.out` and `sso.out` to see how `app1_sso.out` uses the definitions of `main()`, `do_some_work()`, and `xyz` in shared memory.

The disassembly output of `app1_sso.out` shows that references to `main()` from the `libc.a` function `_args_main()` refer to the location where `main()` is defined in shared memory:

```
%> dis7x app1_sso.out app1_sso.dis
%> cat app1_sso.dis
...
0x000070070340             _args_main:
0x000070070340   0000001f   CSTX1: 0x00000000
0x000070070344   000018bd || SUBD .D1 A0,A0,A0
0x000070070348   10001840 || MVKU32 .L1 0,A4
0x00007007034c   0000001f   CSTX1: 0x00000000
0x000070070350   1107914c ||[A0] LDW .D1 *PC(0x10),A4
0x000070070354   201f8772    STD .D2X A8,*D15(0x8)
0x000070070358   0000001f   CSTX1: 0x00000000
0x00007007035c   20016845 || MVC .S1 RP,A8
0x000070070360   14001841 || MVKU32 .L1 0,A5
0x000070070364   400018bc || SUBD .D1 A0,A0,D0
0x000070070368   0000201d   CCEXT0: 0x0000201d
0x00007007036c   148809bd ||[A0] ADDD .D1 D0,0x8,A5
0x000070070370   7d8bc9fd || ADDD .D2 D15,0x18,D15
0x000070070374   ffff80be || CALL .B1 main
...
```

The definitions of `main()`, `do_some_work()`, and `xyz` can be found in the disassembly output of the SSO file:

```
%> dis7x sso.out sso.dis
%> cat sso.dis
...
TEXT Section .shared.text (Little Endian), 0x180 bytes at 0x000070070000

0x000070070100             do_some_work:
0x000070070100   20016845    MVC .S1 RP,A8
0x000070070104   201f8772 || STD .D2X A8,*D15(0x8)
0x000070070108   7d8bc9bc    ADDD .D1 D15,0x18,D15
0x00007007010c   00009f1f    CSTX1: 0x000004f8
0x000070070110   d1079d4c || LDW .D1 *PC(0x9f10),BL0
0x000070070114   00280382    ADDW .L2 BL0,0x2,B0
0x000070070118   00009f1f    CSTX1: 0x000004f8
0x00007007011c   41079750 || STW .D1X B0,*PC(0x9f10)
0x000070070120   00522804    MVC .S1 A8,RP
0x000070070124   202f8d6c    LDD .D1 *D15(0x10),A8
0x000070070128   7c8bc9bd    ADDD .D1 D15,0x8,D15
0x00007007012c   00000c1e || RET .B1

0x000070070140             main:
0x000070070140   20016845    MVC .S1 RP,A8
0x000070070144   201f8772 || STD .D2X A8,*D15(0x8)
0x000070070148   7d8bc9bc    ADDD .D1 D15,0x18,D15
0x00007007014c   fffff0be    CALL .B1 do_some_work
...

DATA Section .shared.data (Little Endian), 0x20 bytes at 0x00007007a010
0x00007007a010             xyz:
0x00007007a010   0000000a           .word 0x0000000a
0x00007007a014   00000014           .word 0x00000014
0x00007007a018   0000001e           .word 0x0000001e
0x00007007a01c   00000028           .word 0x0000001c
```

Returning to the disassembly of `app1_sso.out`, notice that `_args_main` calls `main()`, which resolves to the SSO address of `0x70070140`, which matches the address in the generated `app1.xml` and `app2.xml` link information files:

```
0x000070070340             _args_main:
...
0x000070070374   ffff80be || CALL .B1 main
```


Copyright © 2026 Texas Instruments Incorporated

The do_some_work() function uses the SSO address of xyz (0x7007a010). The core's MMU translates accesses in the shared .shared.data region to each core's own local copy of .shared.data at 0x40001000. When do_some_work() loads from xyz at 0x7007a010, the MMU redirects the access to the core-local address of 0x40001000.

Note

Load and Initialize SSO Content Before Execution: The contents of the SSO file must be loaded and initialized in system memory before executing any code that may access any function or data object defined in the SSO.

Another way to examine the disassembly output is by using the name utility (nm7x) described in Section 13.3. The following command verifies that main and do_some_work resolve to the symbols in the SSO.

```
%> nm7x core1/app1_sso.out | grep -E ' (main|do_some_work|xyz)$'
000070070100 A do_some_work          <- SSO address
000070070140 A main                  <- SSO address
000040001000 B xyz                   <- core-local copy of shared RW data
```

The A type in the output indicates the symbol is resolved from the SSO. The B type indicates a core-local uninitialized or data symbol; the SSO's .shared.data section is duplicated per core and the MMU redirects shared-address accesses to the local copy.

12.12.6 Summary of SSO Command Line Options

For quick reference, these are the command-line options used to create and build the SSO linker command file as described in the previous sections:

--gen_data_subsections=on This compiler option is on by default. It produces per-data-object subsections for structs and arrays. Scalar globals are not placed in subsections even with this option on.

--gen_func_subsections This compiler option is off by default. It produces per-function subsections. Without it, OptiShare has no per-function granularity.

--gen_xml_func_hash When this linker option is combined with the --xml_link_info linker option, the linker includes hash values for functions and data objects and section reference information in the --xml_link_info output.

--import_sso This linker option is used when relinking an application against a previously created SSO file. This option enables an application to execute code and access RO data from shared memory. Sharing these functions and RO data with other applications in the multicore system reduces overall code size.

--sso This linker option instructs the linker to produce an SSO file using a linker command file that specifies the placement of common functions and data objects in shared memory. The linker uses only the SSO linker command file to identify the input sections to include in the link.

--xml_link_info This linker option generates a well-formed XML file containing detailed information about the result of a link.

Copyright © 2026 Texas Instruments Incorporated

Chapter 13

Object File Utilities

This chapter describes how to invoke the following utilities:

- The **object file display utility** prints the contents of object files, executable files, and/or archive libraries in both text and XML formats.
- The **disassembler** accepts object files and executable files as input and produces an assembly listing as output. This listing shows assembly instructions, their opcodes, and the section program counter values.
- The **name utility** prints a list of names defined and referenced in an object file, executable files, and/or archive libraries.
- The **strip utility** removes symbol table and debugging information from object and executable files.

13.1 Invoking the Object File Display Utility 311
13.2 Invoking the Disassembler 312
13.3 Invoking the Name Utility 313
13.4 Invoking the Strip Utility 313

Copyright © 2026 Texas Instruments Incorporated

13.1 Invoking the Object File Display Utility

The object file display utility, *ofd7x*, prints the contents of object files (.obj), executable files (.out), and/or archive libraries (.lib) in both text and XML formats. Hidden symbols are listed as *no name*, while localized symbols are listed like any other local symbol.

To invoke the object file display utility, enter the following:

ofd7x [*options*] *input filename* [*input filename*]

ofd7x is the command that invokes the object file display utility.

input filename names the object file (.obj), executable file (.out), or archive library (.lib) source file. The filename must contain an extension.

options identify the object file display utility options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen.

--call_graph Prints function stack usage and callee information in XML format. While the XML output may be accessed by a developer, this option was primarily designed to be used by tools such as Code Composer Studio to display an application's worst case stack usage.

--dwarf_display=*attributes* Controls DWARF display filter settings by specifying a comma-delimited list of *attributes*. Prefixing an attribute with no disables instead of enables. For example:

```
--dwarf_display=nodabbrev,nodline
--dwarf_display=all,nodabbrev
--dwarf_display=none,dinfo,types
```

The order of attributes is important (see --obj_display). The list of available display attributes can be obtained by invoking ofd7x --dwarf_display=help.

--dwarf Appends DWARF debug information to program output.

--help Displays help

--output=*filename* Sends program output to *filename* rather than to the screen.

--obj_display *attributes* Controls object file display filter settings by specifying a comma-delimited list of *attributes*. Prefixing an attribute with no disables instead of enables. For example:

```
--obj_display=rawdata,nostrings
--obj_display=all,norawdata
--obj_display=none,header
```

The order of attributes is important. For instance, in "--obj_display=none,header", ofd7x disables all output, then re-enables file header information. If the attributes are specified in the reverse order, (header,none), the file header is enabled, then all output is disabled, including the file header. Thus, nothing is printed to the screen for the given files. The list of available display attributes can be obtained by invoking ofd7x --obj_display=help.

--verbose Prints verbose text output.

--xml Displays output in XML format.

--xml_indent=*num* Sets the number of spaces to indent nested XML tags.

If an archive file is given as input to the object file display utility, each object file member of the archive is processed as if it was passed on the command line. The object file members are processed in the order in which they appear in the archive file.

If the object file display utility is invoked without any options, it displays information about the contents of the input files on the console screen.

Note

Object File Display Format: The object file display utility produces data in a text format by default. This data is not intended to be used as input to programs for further processing of the information. XML format should be used for mechanical processing.

Copyright © 2026 Texas Instruments Incorporated

13.2 Invoking the Disassembler

The disassembler, *dis7x*, examines the output of the assembler or linker. This utility accepts an object file or executable file as input and writes the disassembled object code to standard output or a specified file.

To invoke the disassembler, enter the following:

dis7x[*options*] *input filename*[.] [*output filename*]

dis7x is the command that invokes the disassembler.

options identifies the name utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (-). The name utility options are as follows:

--all (-1)	disassembles all sections, processes .cinit sections
--noaddr (-a)	disables the printing of branch destination addresses along with labels.
--bytes (-b)	displays data as bytes instead of words.
-c	dumps the object file information.
--nodata (-d)	disables display of data sections.
--hex (-e)	displays integer values in hexadecimal.
--help (-h)	shows the current help screen.
--data_as_text (-i)	disassembles .data sections as instructions.
--text_as_data (-l)	disassembles data sections as text.
--loadtime_addr (-L)	displays both load and run addresses if they are different.
--single_opcode (-o)##	disassembles single word ## or 0x## then exits.
--quiet (-q)	(quiet mode) suppresses the banner and all progress information.
--realquiet (-qq)	(super quiet mode) suppresses all headers.
--suppress (-s)	suppresses printing of address and data words.
--notext (-t)	suppresses the display of text sections in the listing.
--silicon_version (-v)	displays family of the target.
--copy_tables (-y)	displays copy tables and the sections copied. The table information is dumped first, then each record followed by its load and run data.

input filename[*.ext*] This is the name of the input file. If the optional extension is not specified, the file is searched for in this order:

1. *infile*
2. *infile*.out, an executable file
3. *infile*.obj, an object file

output filename is the name of the optional output file to which the disassembly will be written. If an output filename is not specified, the disassembly is written to standard output.

Copyright © 2026 Texas Instruments Incorporated

13.3 Invoking the Name Utility

The name utility, *nm7x*, prints the list of names defined and referenced in an object file, executable file, or archive library. It also prints the symbol value and an indication of the kind of symbol. Hidden symbols are listed with an empty string ("") as their name. To invoke the name utility, enter the following:

nm7x [*-options*] [*input filenames*]

nm7x	is the command that invokes the name utility.
input filename	is an object file (.obj), executable file (.out), or archive library (.lib).
options	identifies the name utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (-). The name utility options are as follows:

--all (-a)	prints all symbols.
--prep_fname (-f)	prepends file name to each symbol.
--global (-g)	prints only global symbols.
--help (-h)	shows the current help screen.
--format:long (-l)	produces a detailed listing of the symbol information.
--sort:value (-n)	sorts symbols numerically rather than alphabetically.
--output (-o) *file*	outputs to the given file.
--sort:none (-p)	causes the name utility to not sort any symbols.
--quiet (-q)	(quiet mode) suppresses the banner and all progress information.
--sort:reverse (-r)	sorts symbols in reverse order.
--dynamic (-s)	lists symbols in the dynamic symbol table for an ELF object module.
--undefined (-u)	only prints undefined symbols.

13.4 Invoking the Strip Utility

The strip utility, *strip7x*, removes symbol table and debugging information from object and executable files. To invoke the strip utility, enter the following:

strip7x [**-p**] *input filename* [*input filename*]

strip7x	is the command that invokes the strip utility.
input filename	is an object file (.obj) or an executable file (.out).
options	identifies the strip utility options you want to use. Options are not case sensitive and can appear anywhere on the command line following the invocation. Precede each option with a hyphen (-). The strip utility option is as follows:

--help (-h)	displays help information.
--outfile (-o) *filename*	writes the stripped output to filename.
--postlink (-p)	removes all information not required for execution. This option causes more information to be removed than the default behavior, but the object file is left in a state that cannot be linked. This option should be used only with static executable files.
--rom	Strips read-only sections and segments.

When the strip utility is invoked without the -o option, the input object files are replaced with the stripped version.

Copyright © 2026 Texas Instruments Incorporated

Chapter 14

C++ Name Demangler

The C++ compiler implements function overloading, operator overloading, and type-safe linking by encoding a function's prototype and namespace in its link-level name. The process of encoding the prototype into the linkname is often referred to as name mangling. When you inspect mangled names, such as in assembly files, disassembler output, or compiler or linker diagnostic messages, it can be difficult to associate a mangled name with its corresponding name in the C++ source code. The C++ name demangler is a debugging aid that translates each mangled name it detects to its original name found in the C++ source code.

These topics tell you how to invoke and use the C++ name demangler. The C++ name demangler reads in input, looking for mangled names. All unmangled text is copied to output unaltered. All mangled names are demangled before being copied to output.

14.1 Invoking the C++ Name Demangler 315
14.2 Sample Usage of the C++ Name Demangler 316

Copyright © 2026 Texas Instruments Incorporated

14.1 Invoking the C++ Name Demangler

The syntax for invoking the C++ name demangler is:

dem7x [*options*] [*filenames*]

dem7x	Command that invokes the C++ name demangler.
options	Options affect how the name demangler behaves. Options can appear anywhere on the command line.
filenames	Text input files, such as the assembly file output by the compiler, the assembler listing file, the disassembly file, and the linker map file. If no filenames are specified on the command line, dem7x uses standard input.

By default, the C++ name demangler outputs to standard output. You can use the -o file option if you want to output to a file.

The following options apply only to the C++ name demangler:

--debug (--d)	Prints debug messages.
--diag_wrap[=on,off]	Sets diagnostic messages to wrap at 79 columns (on, which is the default) or not (off).
--help (-h)	Prints a help screen that provides an online summary of the C++ name demangler options.
--output= *file* (-o)	Outputs to the specified file rather than to standard out.
--quiet (-q)	Reduces the number of messages generated during execution.

Copyright © 2026 Texas Instruments Incorporated

14.2 Sample Usage of the C++ Name Demangler

The examples in this section illustrate the demangling process.

This example shows a sample C++ program. In this example, the linknames of all the functions are mangled; that is, their signature information is encoded into their names.

```
class banana {
public:
    int calories(void);
    banana();
    ~banana();
};
int calories_in_a_banana(void)
{
    banana x;
    return x.calories();
}
```

Executing the C++ name demangler will demangle all names that it believes to be mangled. Enter:

```
dem7x calories_in_a_banana.asm
```

The result after running the C++ name demangler is as follows. The linknames _ZN6bananaC1Ev, _ZN6banana8caloriesEv, and _ZN6bananaD1Ev are demangled.

```
||calories_in_a_banana()||:
;** --------------------------------------------------------------------------*
           MVC     .S1     RP,A9             ; [A_S1]
||         STD     .D1     A8,*SP(8)         ; [A_D1]
||         STD     .D2X    A9,*SP++(-24)     ; [A_D2]
           CALL    .B1     ||banana::banana()|| ; [A_B] |9|
||         ADDD    .D1     SP,0x10,A4        ; [A_D1] |9|
$C$RL0:    ; CALL OCCURS (||banana::banana()||) arg:{A4} ret:{}  ; [] |9|
           CALL    .B1     ||banana::calories()|| ; [A_B] |10|
||         ADDD    .D1     SP,0x10,A4        ; [A_D1] |10|
$C$RL1:    ; CALL OCCURS (||banana::calories()||) arg:{A4} ret:{A4}  ; [] |10|
           CALL    .B1     ||banana::~banana()|| ; [A_B] |10|
||         ADDD    .D1     SP,0x10,A4        ; [A_D1] |10|
||         MV      .D2     A4,A8             ; [A_D2] |10|
$C$RL2:    ; CALL OCCURS (||banana::~banana()||) arg:{A4} ret:{}  ; [] |10|
           MV      .D1     A8,A4             ; [A_D1] |10|
           MVC     .S1     A9,RP             ; [A_S1] BARRIER
           LDD     .D1     *SP(24),A9        ; [A_D1]
||         LDD     .D2     *SP(32),A8        ; [A_D2]
           RET     .B1     ; [A_B]
||         ADDD    .D1     SP,0x18,SP        ; [A_D1]
           ; RETURN OCCURS {RP}              ; []
```


Copyright © 2026 Texas Instruments Incorporated

Appendix A

XML Link Information File Description

The C7000 linker supports the generation of an XML link information file via the --xml_link_info *file* option. This option causes the linker to generate a well-formed XML file containing detailed information about the result of a link. The information included in this file includes all of the information that is currently produced in a linker-generated map file.

As the linker evolves, the XML link information file may be extended to include additional information that could be useful for static analysis of linker results.

This appendix enumerates all of the elements that are generated by the linker into the XML link information file.

A.1 XML Information File Element Types 318
A.2 Document Elements 318

Copyright © 2026 Texas Instruments Incorporated

A.1 XML Information File Element Types

These element types will be generated by the linker:

- **Container elements** represent an object that contains other elements that describe the object. Container elements have an id attribute that makes them accessible from other elements.
- **String elements** contain a string representation of their value.
- **Constant elements** contain a 64-bit unsigned long representation of their value (with a 0x prefix).
- **Reference elements** are empty elements that contain an idref attribute that specifies a link to another container element.

In Section A.2, the element type is specified for each element in parentheses following the element description. For instance, the <link_time> element lists the time of the link execution (string).

A.2 Document Elements

The root element, or the document element, is **<link_info>**. All other elements contained in the XML link information file are children of the <link_info> element. The following sections describe the elements that an XML information file can contain.

A.2.1 Header Elements

The first elements in the XML link information file provide general information about the linker and the link session:

- The **<banner>** element lists the name of the executable and the version information (string).
- The **<copyright>** element lists the TI copyright information (string).
- The **<link_time>** is a timestamp representation of the link time (unsigned 32-bit int).
- The **<output_file>** element lists the absolute path and name of the linked output file generated (string).
- The **<entry_point>** element specifies the program entry point, as determined by the linker (container) with two entries:
  - The **<name>** is the entry point symbol name, if any (string).
  - The **<address>** is the entry point address (constant).

Header Element for the hi.out Output File

```
<banner>TMS320Cxx Linker          Version x.xx (Jan 6 2008)</banner>
<copyright>Copyright (c) 1996-2008 Texas Instruments Incorporated</copyright>
<link_time>0x43dfd8a4</link_time>
<output_file>/usr/mycode/hi.out</output_file>
<entry_point>
   <name>_c_int00</name>
   <address>0xaf80</address>
</entry_point>
```


Copyright © 2026 Texas Instruments Incorporated

A.2.2 Input File List

The next section of the XML link information file is the input file list, which is delimited with a **<input_file_list>** container element. The <input_file_list> can contain any number of <input_file> elements.

Each **<input_file>** instance specifies the input file involved in the link. Each <input_file> has an id attribute that can be referenced by other elements, such as an <object_component>. An <input_file> is a container element enclosing the following elements:

- The **<path>** element names an absolute directory path, if applicable (string).
- The **<kind>** element specifies a file type, either archive or object (string).
- The **<file>** element specifies an archive name or filename (string).
- The **<name>** element specifies an object file name, or archive member name (string).

Input File List for the hi.out Output File

```
<input_file_list>
   <input_file id="fl-1">
      <kind>object</kind>
      <file>hi.obj</file>
      <name>hi.obj</name>
   </input_file>
   <input_file id="fl-2">
      <path>/usr/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>boot.obj</name>
   </input_file>
   <input_file id="fl-3">
      <path>/usr/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>exit.obj</name>
   </input_file>
   <input_file id="fl-4">
      <path>/usr/tools/lib/</path>
      <kind>archive</kind>
      <file>rtsxxx.lib</file>
      <name>printf.obj</name>
   </input_file>
...
</input_file_list>
```


Copyright © 2026 Texas Instruments Incorporated

A.2.3 Object Component List

The next section of the XML link information file contains a specification of all of the object components that are involved in the link. An example of an object component is an input section. In general, an object component is the smallest piece of object that can be manipulated by the linker.

The **<object_component_list>** is a container element enclosing any number of <object_component> elements.

Each **<object_component>** specifies a single object component. Each <object_component> has an id attribute so that it can be referenced directly from other elements, such as a <logical_group>. An <object_component> is a container element enclosing the following elements:

- The **<name>** element names the object component (string).
- The **<load_address>** element specifies the load-time address of the object component (constant).
- The **<run_address>** element specifies the run-time address of the object component (constant).
- The **<alignment>** element specifies the alignment of the object component (unsigned int).
- The **<size>** element specifies the size of the object component (constant).
- The **<executable>** element specifies whether the object component allows execute access (string). If an object has executable access, it cannot also have read-write access. While "false" is a valid value, the linker emits this element only if it is "true" for this object component.
- The **<readonly>** element specifies whether the object component allows read-only access (string). If an object has read-only access, it cannot also have read-write access. While "false" is a valid value, the linker emits this element only if it is "true" for this object component.
- The **<readwrite>** element specifies whether the object component allows read-write access (string). If an object has read-write access, it cannot also have read-only access or executable access. While "false" is a valid value, the linker emits this element only if it is "true" for this object component.
- The **<uninitialized>** element specifies whether the object component is initialized (string). While "false" is a valid value, the linker emits this element only if it is "true" for this object component.
- The **<input_file_ref>** element specifies the source file where the object component originated (reference).

Object Component List for the fl-4 Input File

```
<object_component id="oc-20">
   <name>.text</name>
   <load_address>0xac00</load_address>
   <run_address>0xac00</run_address>
   <alignment>0x1</alignment>
   <size>0xc0</size>
   <readonly>true</readonly>
   <executable>false</executable>
   <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-21">
   <name>.data</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <alignment>0x1</alignment>
   <size>0x0</size>
   <readwrite>true</readwrite>
   <input_file_ref idref="fl-4"/>
</object_component>
<object_component id="oc-22">
   <name>.bss</name>
   <load_address>0x80000000</load_address>
   <run_address>0x80000000</run_address>
   <alignment>0x1</alignment>
   <size>0x0</size>
   <readwrite>true</readwrite>
   <uninitialized>true</uninitialized>
   <input_file_ref idref="fl-4"/>
</object_component>
```


Copyright © 2026 Texas Instruments Incorporated

A.2.4 Logical Group List

The **<logical_group_list>** section of the XML link information file is similar to the output section listing in a linker-generated map file. However, the XML link information file contains a specification of GROUP and UNION output sections, which are not represented in a map file. There are several kinds of list items that can occur in a <logical_group_list>:

- The **<logical_group>** is the specification of a section or GROUP that contains a list of object components or logical group members. Each <logical_group> element is given an id so that it may be referenced from other elements. Each <logical_group> is a container element enclosing the following elements:
  - The **<name>** element names the logical group (string).
  - The **<load_address>** element specifies the load-time address of the logical group (constant).
  - The **<run_address>** element specifies the run-time address of the logical group (constant).
  - The **<size>** element specifies the size of the logical group (constant).
  - The **<output_section_group>** specifies whether the logical group is a GROUP of output sections (string). While "false" is a valid value, the linker emits this element only if it is "true" for this logical group.
  - The **<contents>** element lists elements contained in this logical group (container). These elements refer to each of the member objects contained in this logical group:
    - The **<object_component_ref>** is an object component that is contained in this logical group (reference).
    - The **<logical_group_ref>** is a logical group that is contained in this logical group (reference).
- The **<overlay>** is a special kind of logical group that represents a UNION, or a set of objects that share the same memory space (container). Each <overlay> element is given an id so that it may be referenced from other elements (like from an <allocated_space> element in the placement map). Each <overlay> contains the following elements:
  - The **<name>** element names the overlay (string).
  - The **<run_address>** element specifies the run-time address of the overlay (constant).
  - The **<size>** element specifies the size of the logical group (constant).
  - The **<contents>** container element lists elements contained in this overlay. These elements refer to each of the member objects contained in this logical group:
    - The **<object_component_ref>** is an object component that is contained in this logical group (reference).
    - The **<logical_group_ref>** is a logical group that is contained in this logical group (reference).
- The **<split_section>** is another special kind of logical group that represents a collection of logical groups that is split among multiple memory areas. Each <split_section> element is given an id so that it may be referenced from other elements. The id consists of the following elements.
  - The **<name>** element names the split section (string).
  - The **<contents>** container element lists elements contained in this split section. The <logical_group_ref> elements refer to each of the member objects contained in this split section, and each element referenced is a logical group that is contained in this split section (reference).

Copyright © 2026 Texas Instruments Incorporated

Logical Group List for the fl-4 Input File

```
<logical_group_list>
   ...
   <logical_group id="lg-7">
      <name>.text</name>
      <output_section_group>true</output_section_group>
      <load_address>0x20</load_address>
      <run_address>0x20</run_address>
      <size>0xb240</size>
      <contents>
         <object_component_ref idref="oc-34"/>
         <object_component_ref idref="oc-108"/>
         <object_component_ref idref="oc-e2"/>
         ...
      </contents>
   </logical_group>
   ...
   <overlay id="lg-b">
      <name>UNION_1</name>
      <run_address>0xb600</run_address>
      <size>0xc0</size>
      <contents>
         <object_component_ref idref="oc-45"/>
         <logical_group_ref idref="lg-8"/>
      </contents>
   </overlay>
   ...
   <split_section id="lg-12">
      <name>.task_scn</name>
      <size>0x120</size>
      <contents>
         <logical_group_ref idref="lg-10"/>
         <logical_group_ref idref="lg-11"/>
      </contents>
   </split_section>
   ...
</logical_group_list>
```


Copyright © 2026 Texas Instruments Incorporated

A.2.5 Placement Map

The **<placement_map>** element describes the memory placement details of all named memory areas in the application, including unused spaces between logical groups that have been placed in a particular memory area.

The **<memory_area>** is a description of the placement details within a named memory area (container). The description consists of these items:

- The **<name>** names the memory area (string).
- The **<page_id>** gives the id of the memory page in which this memory area is defined (constant).
- The **<origin>** specifies the beginning address of the memory area (constant).
- The **<length>** specifies the length of the memory area (constant).
- The **<used_space>** specifies the amount of allocated space in this area (constant).
- The **<unused_space>** specifies the amount of available space in this area (constant).
- The **<attributes>** lists the RWXI attributes that are associated with this area, if any (string).
- The **<fill_value>** specifies the fill value that is to be placed in unused space, if the fill directive is specified with the memory area (constant).
- The **<usage_details>** lists details of each allocated or available fragment in this memory area. If the fragment is allocated to a logical group, then a <logical_group_ref> element is provided to facilitate access to the details of that logical group. All fragment specifications include <start_address> and <size> elements.
  - The **<allocated_space>** element provides details of an allocated fragment within this memory area (container):
    - The **<start_address>** specifies the address of the fragment (constant).
    - The **<size>** specifies the size of the fragment (constant).
    - The **<logical_group_ref>** provides a reference to the logical group that is allocated to this fragment (reference).
  - The **<available_space>** element provides details of an available fragment within this memory area (container):
    - The **<start_address>** specifies the address of the fragment (constant).
    - The **<size>** specifies the size of the fragment (constant).

Placement Map for the fl-4 Input File

```
<placement_map>
   <memory_area>
      <name>PMEM</name>
      <page_id>0x0</page_id>
      <origin>0x20</origin>
      <length>0x100000</length>
      <used_space>0xb240</used_space>
      <unused_space>0xf4dc0</unused_space>
      <attributes>RWXI</attributes>
      <usage_details>
         <allocated_space>
            <start_address>0x20</start_address>
            <size>0xb240</size>
            <logical_group_ref idref="lg-7"/>
         </allocated_space>
         <available_space>
            <start_address>0xb260</start_address>
            <size>0xf4dc0</size>
         </available_space>
      </usage_details>
   </memory_area>
   ...
</placement_map>
```


Copyright © 2026 Texas Instruments Incorporated

A.2.6 Far Call Trampoline List

The **<far_call_trampoline_list>** is a list of <far_call_trampoline> elements. The linker supports the generation of far call trampolines to help a call site reach a destination that is out of range. A far call trampoline function is guaranteed to reach the called function (callee) as it may utilize an indirect call to the called function.

The <far_call_trampoline_list> enumerates all of the far call trampolines that are generated by the linker for a particular link. The <far_call_trampoline_list> can contain any number of <far_call_trampoline> elements. Each **<far_call_trampoline>** is a container enclosing the following elements:

- The **<callee_name>** element names the destination function (string).
- The **<callee_address>** is the address of the called function (constant).
- The **<trampoline_object_component_ref>** is a reference to an object component that contains the definition of the trampoline function (reference).
- The **<trampoline_address>** is the address of the trampoline function (constant).
- The **<caller_list>** enumerates all call sites that utilize this trampoline to reach the called function (container).
- The **<trampoline_call_site>** provides the details of a trampoline call site (container) and consists of these items:
  - The **<caller_address>** specifies the call site address (constant).
  - The **<caller_object_component_ref>** is the object component where the call site resides (reference).

Far Call Trampoline List for the fl-4 Input File

```
<far_call_trampoline_list>
   ...
   <far_call_trampoline>
      <callee_name>_foo</callee_name>
      <callee_address>0x08000030</callee_address>
      <trampoline_object_component_ref idref="oc-123"/>
      <trampoline_address>0x2020</trampoline_address>
      <caller_list>
         <call_site>
            <caller_address>0x1800</caller_address>
            <caller_object_component_ref idref="oc-23"/>
         </call_site>
         <call_site>
            <caller_address>0x1810</caller_address>
            <caller_object_component_ref idref="oc-23"/>
         </call_site>
      </caller_list>
   </far_call_trampoline>
   ...
</far_call_trampoline_list>
```


Copyright © 2026 Texas Instruments Incorporated

A.2.7 Symbol Table

The **<symbol_table>** contains a list of all of the global symbols that are included in the link. The list provides information about a symbol's name and value. In the future, the symbol_table list may provide type information, the object component in which the symbol is defined, storage class, etc.

The **<symbol>** is a container element that specifies the name and value of a symbol with these elements:

- The **<name>** element specifies the symbol name (string).
- The **<value>** element specifies the symbol value (constant).
- The **<local>** element specifies whether the symbol has local binding (string). While "false" is a valid value, the linker emits this element only if it is "true" for this symbol.

Symbol Table for the fl-4 Input File

```
<symbol_table>
   <symbol>
      <name>_c_int00</name>
      <value>0xaf80</value>
   </symbol>
   <symbol>
      <name>_main</name>
      <value>0xb1e0</value>
   </symbol>
   <symbol>
      <name>_printf</name>
      <value>0xac00</value>
      <local>true</local>
   </symbol>
   ...
</symbol_table>
```


Copyright © 2026 Texas Instruments Incorporated

Appendix B

Unsupported Tools and Features

B.1 List of Unsupported Tools and Features

The following tools and features will not be supported by the C7000 compiler toolset:

- Compiler Consultant
- Profile Directed Compilation
- CCS Optimizer Assistant
- Cache Layout Tool
- Hex Utility (We recommend that you instead use third-party tools, such as GNU “objcopy” or ARM Ltd. Tools.)
- Cross Reference Tool
- Absolute Lister
- C6x Linear Assembly
- C7x Linear Assembly
- C6x (Legacy) Assembly Compatibility
- MISRA-C:2004 and MISRA-C:2012 checking (We recommend that you use third-party tooling, such as LDRA.)

Copyright © 2026 Texas Instruments Incorporated

Appendix C

Glossary

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

aliasing The ability for a single object to be accessed in more than one way, such as when two pointers point to a single object. It can disrupt optimization, because any indirect reference could refer to any other object.

allocation A process in which the linker calculates the final memory addresses of output sections.

ANSI American National Standards Institute; an organization that establishes standards voluntarily followed by industries.

Application Binary Interface (ABI) A standard that specifies the interface between two object modules. An ABI specifies how functions are called and how information is passed from one program component to another.

archive library A collection of individual files grouped into a single file by the archiver.

archiver A software program that collects several individual files into a single file called an archive library. With the archiver, you can add, delete, extract, or replace members of the archive library.

assembler A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro definitions. The assembler substitutes absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

assignment statement A statement that initializes a variable with a value.

autoinitialization The process of initializing global C variables (contained in the .cinit section) before program execution begins.

autoinitialization at run time An autoinitialization method used by the linker when linking C code. The linker uses this method when you invoke it with the --rom_model link option. The linker loads the .cinit section of data tables into memory, and variables are initialized at run time.

big endian An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *little endian*

block A set of statements that are grouped together within braces and treated as an entity.

Copyright © 2026 Texas Instruments Incorporated

.bss section One of the default object file sections. You use the assembler .bss directive to reserve a specified amount of space in the memory map that you can use later for storing data. The .bss section is uninitialized.

byte Per ANSI/ISO C, the smallest addressable unit that can hold a character.

C/C++ compiler A software program that translates C source statements into assembly language source statements.

code generator A compiler tool that takes the file produced by the parser or the optimizer and produces an assembly language source file.

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.

complex type A complex scalar type. For example, cint and cfloat.

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.

element type The element type of an array is the type of each element in the array. The element type of a complex type is the type of each component of the complex type.

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 device's operation.

Copyright © 2026 Texas Instruments Incorporated

entry point	A point in target memory where execution starts.
environment variable	A system symbol that you define and assign to a string. Environmental variables are often included in Windows batch files or UNIX shell scripts such as .cshrc or .profile.
epilog	The portion of code in a function that restores the stack and returns.
executable object file	A linked, executable object file that is downloaded and executed on a target system.
expression	A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.
external symbol	A symbol that is used in the current program module but defined or declared in a different program module.
file-level optimization	A level of optimization where the compiler uses the information that it has about the entire file to optimize your code (as opposed to program-level optimization, where the compiler uses information that it has about the entire program to optimize your code).
function inlining	The process of inserting code for a function at the point of call. This saves the overhead of a function call and allows the optimizer to optimize the function in the context of the surrounding code.
global symbol	A symbol that is either defined in the current module and accessed in another, or accessed in the current module but defined in another.
high-level language debugging	The ability of a compiler to retain symbolic and high-level language information (such as type and function definitions) so that a debugging tool can use this information.
indirect call	A function call where one function calls another function by giving the address of the called function.
initialization at load time	An autoinitialization method used by the linker when linking C/C++ code. The linker uses this method when you invoke it with the --ram_model link option. This method initializes variables at load time instead of run time.
initialized section	A section from an object file that will be linked into an executable object file.
input section	A section from an object file that will be linked into an executable object file.
integrated preprocessor	A C/C++ preprocessor that is merged with the parser, allowing for faster compilation. Stand-alone preprocessing or preprocessed listing is also available.
interlist feature	A feature that inserts as comments your original C/C++ source statements into the assembly language output from the assembler. The C/C++ statements are inserted next to the equivalent assembly instructions.
intrinsics	Operators that are used like functions and produce assembly language code that would otherwise be inexpressible in C, or would take greater time and effort to code.
ISO	International Organization for Standardization; a worldwide federation of national standards bodies, which establishes international standards voluntarily followed by industries.

Copyright © 2026 Texas Instruments Incorporated

iteration count The number of times that a loop executes before it terminates. Previously called the "trip count".

kernel The body of a software-pipelined loop between the pipelined-loop prolog and the pipelined-loop epilog.

K&R C Kernighan and Ritchie C, the de facto standard as defined in the first edition of *The C Programming Language* (K&R). Most K&R C programs written for earlier, non-ISO C compilers should correctly compile and run without modification.

label A symbol that begins in column 1 of an assembler source statement and corresponds to the address of that statement. A label is the only assembler statement that can begin in column 1.

linker A software program that combines object files to form an executable object file that can be allocated into system memory and executed by the device.

listing file An output file, created by the assembler, which lists source statements, their line numbers, and their effects on the section program counter (SPC).

little endian An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is hardware-specific and is determined at reset. See also *big endian*

loader A device that places an executable object file into system memory.

loop unrolling An optimization that expands small loops so that each iteration of the loop appears in your code. Although loop unrolling increases code size, it can improve the performance of your code.

macro A user-defined routine that can be used as an instruction.

macro call The process of invoking a macro.

macro definition A block of source statements that define the name and the code that make up a macro.

macro expansion The process of inserting source statements into your code in place of a macro call.

map file An output file, created by the linker, which shows the memory configuration, section composition, section allocation, symbol definitions and the addresses at which the symbols were defined for your program.

memory map A map of target system memory space that is partitioned into functional blocks.

name mangling A compiler-specific feature that encodes a function name with information regarding the function's arguments return types.

object file An assembled or linked file that contains machine-language object code.

object library An archive library made up of individual object files.

Copyright © 2026 Texas Instruments Incorporated

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.

pipelining A technique where a second instruction begins executing before the first instruction has been completed. You can have several instructions in the pipeline, each at a different processing stage.

pop An operation that retrieves a data object from a stack.

pragma A preprocessor directive that provides directions to the compiler about how to treat a particular statement.

preprocessor A software tool that interprets macro definitions, expands macros, interprets header files, interprets conditional compilation, and acts upon preprocessor directives.

program-level optimization An aggressive level of optimization where all of the source files are compiled into one intermediate file. Because the compiler can see the entire program, several optimizations are performed with program-level optimization that are rarely applied during file-level optimization.

prolog The portion of code in a function that sets up the stack.

push An operation that places a data object on a stack for temporary storage.

quiet run An option that suppresses the normal banner and the progress information.

raw data Executable code or initialized data in an output section.

relocation A process in which the linker adjusts all the references to a symbol when the symbol's address changes.

run-time environment The run time parameters in which your program must function. These parameters are defined by the memory and register conventions, stack organization, function call conventions, and system initialization.

run-time-support functions Standard ISO functions that perform tasks that are not part of the C language (such as memory allocation, string conversion, and string searches).

Copyright © 2026 Texas Instruments Incorporated

run-time-support library	A library file, rts.src, which contains the source for the run time-support functions.
scalar type	Any non-aggregate type such as an integer, floating-point, complex, or pointer type.
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.
Single Instruction, Multiple Data (SIMD)	A SIMD operation takes one or more vectors of the same type and performs the same operation on every element in parallel. Not every vector operation is a SIMD operation.
software pipelining	A technique used by the C/C++ optimizer to schedule instructions from a loop so that multiple iterations of the loop execute in parallel.
source file	A file that contains C/C++ code or assembly language code that is compiled or assembled to form an object file.
stand-alone preprocessor	A software tool that expands macros, #include files, and conditional compilation as an independent program. It also performs integrated preprocessing, which includes parsing of instructions.
static variable	A variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; their previous value is resumed when the function or program is reentered.
storage class	An entry in the symbol table that indicates how to access a symbol.
string table	A table that stores symbol names that are longer than eight characters (symbol names of eight characters or longer cannot be stored in the symbol table; instead they are stored in the string table). The name portion of the symbol's entry points to the location of the string in the string table.
subsection	A relocatable block of code or data that ultimately will occupy continuous space in the memory map. Subsections are smaller sections within larger sections. Subsections give you tighter control of the memory map.
symbol	A string of alphanumeric characters that represents an address or a value.
symbolic debugging	The ability of a software tool to retain symbolic information that can be used by a debugging tool such as an emulator.
target system	The system on which the object code you have developed is executed.
.text section	One of the default object file sections. The .text section is initialized and contains executable code. You can use the .text directive to assemble code into the .text section.
trigraph sequence	A 3-character sequence that has a meaning (as defined by the ISO 646-1983 Invariant Code Set). These characters cannot be represented in the C character set and are expanded to one character. For example, the trigraph ??' is expanded to ^.

Copyright © 2026 Texas Instruments Incorporated

unconfigured memory Memory that is not defined as part of the memory map and cannot be loaded with code or data.

uninitialized section A object file section that reserves space in the memory map but that has no actual contents.

unsigned value A value that is treated as a nonnegative number, regardless of its actual sign.

variable A symbol representing a quantity that can assume any of a set of values.

vector type Similar to an array type, but supports arithmetic operations. A vector type is considered an aggregate type.

word A 32-bit addressable location in target memory

Copyright © 2026 Texas Instruments Incorporated

Revision History

Changes from March 14, 2025 to November 20, 2026 (from Revision K (March 2025) to Revision L (November 2026)) **Page**

- Added the --deprecated_logical_vector_semantics command-line option.... 24
- Added the --silicon_errata_i2528 command-line option.... 24
- Added the --silicon_errata_i2532 command-line option.... 24
- Added documentation for the --honor_cmdfile_order linker option.... 30
- Added documentation for the --start-group and --end-group linker options.... 30
- Added 3_1024 as an option for the --mma_version command-line option.... 34
- Added the --silicon_errata_i2528 command-line option.... 34
- Added the --silicon_errata_i2532 command-line option.... 34
- Added the --silicon_version=7524 and =7604 command-line options.... 35
- Documented which file types are placed according to the --temp_directory option.... 37
- Documented which file types are placed according to the --temp_directory option.... 39
- Added macro for new --mma_version setting and vector size settings for C7604 devices.... 39
- Added a predefined macro name for C7604 devices.... 39
- Updated software pipelining default output.... 63
- Updated lists of resource partition units and resource bound units.... 63
- Documented target-specific vector width for SE data.... 75
- Documented fp8_e4m3_ and bf16_ types and normal vs. subnormal minimum values.... 93
- Documented TI syntax complex types.... 93
- Documented that enum types smaller than an int are not supported.... 94
- Documented fp8_e4m3_ and bf16_ types.... 95
- Updated vector data type sizes supported to include 128 element vectors.... 95
- Document vector widths for current C7000 devices.... 95
- Documented complex types in addition to the complex vector types that were documented previously.... 95
- Cross-referenced C99 and TI syntax for complex variables and clarified role of complex.h.... 99
- Allowed for larger C7604 larger vector widths.... 102
- Added fp8_e4m3_ and bf16_ types.... 102
- Added information about fp8_e4m3_ and bf16_ vectors.... 136
- Updated unary and binary operator tables to include Boolean vector support.... 137
- Added information about fp8_e4m3_ and bf16_ vector operators.... 137
- Add types of vectors legal with each operator and details about unary & operator for vectors.... 137
- Add types of vectors legal with each operator.... 137
- Ternary operators are supported for vectors without manual optimization requirements.... 142
- Provides additional examples and details for vector swizzle operators.... 142
- Mention differing vector predicate widths.... 145
- Allowed for larger C7604 larger vector widths.... 146
- Boolean vectors now support operators.... 146
- Described new vs. old logical and Boolean vector operations.... 147
- Added "evec" scalable vector types.... 151
- Removed mention of the .TI.crctab section, which is not used for C7000.... 155
- Mention differing vector widths.... 165

Copyright © 2026 Texas Instruments Incorporated

- Size for passing large arguments by reference depends on target vector width....167
- Size for passing structure arguments by value depends on target vector width....168
- Removed mention of the .TI.crctab section, which is not used for C7000....223
- Added documentation for the --start-group and --end-group linker options....228
- Added documentation for the --honor_cmdfile_order linker option....228
- Added predefined macro names for C7524 and C7604 devices....233
- Added documentation for the --start-group and --end-group linker options....238
- Document that vector predicated stores extend to a target-dependent size....297
- Added documentation for OptiShare, a multicore memory-sharing feature....301
- Provided an example for OptiShare....302

Changes from March 15, 2024 to March 14, 2025 (from Revision J (March 2024) to Revision K (March 2025)) **Page**

- Removed references to SPRUIG4, which is no longer available....15
- Added 2_256f as a setting for the --mma_version option....24
- Added 2_256f as a setting for the --mma_version option....34
- Added predefined macro names for C7524 devices, MMA support, Streaming Engines, and Streaming Address Generators....39
- Revised explanation of availability of standard and added keywords in strict ANSI mode....98
- Clarified information about using the CLINK pragma, since conditional linking is enabled by default....107
- The full set of library functions and macros declared in math.h is supported....128
- Added an example that initializes a complex vector....136
- Add documentation for unary & operator for vectors....137
- Added documentation for use of vector pointers....138
- The classes provided by the c7x_scalable.h header file use the TI vector types....151
- Added description of c7x_iluthist.h internal header file....177
- Clarified information about using the CLINK pragma, since conditional linking is enabled by default....220
- Added terms related to vectors and complex types....327

Changes from December 15, 2023 to March 15, 2024 (from Revision I (December 2023) to Revision J (March 2024)) **Page**

- Corrected syntax for compiler options related to silicon errata....24
- Corrected syntax for compiler options related to silicon errata....34
- Added that Streaming Engines are most efficient when reading from L2....75
- Added section on Theory of Operation for Streaming Engines and Streaming Address Generators....75
- Specified iteration counter and dimension size offset more specifically....76
- Improved descriptions of SE and SA interfaces....77
- Clarified that dimension offsets, not dimension sizes, are used for SE and SA....77
- Added note about the types that may be passed to SE APIs....79
- Added note about the types that may be passed to SA APIs....80
- The --rtti option is no longer documented. RTTI generation is always enabled for C++ applications....92
- Clarified information about using the CLINK pragma, since conditional linking is enabled by default....107
- The __C7X_UNSTABLE_API macro no longer needs to be defined to access the MMA scalable vector programming utilities....151
- Added information about time zones in RTS functions....179
- Clarified information about using the CLINK pragma, since conditional linking is enabled by default....220
- The <output_file> element in the <link_info> now also includes the absolute path to the output file....318
- The <path> element in the <input_file> now consistently shows the absolute path....319

Copyright © 2026 Texas Instruments Incorporated

- An <object_component> can now also contain <alignment>, <readonly>, <readwrite>, <executable>, and <uninitialized>.....320
- A <logical_group> can now contain an <output_section_group> flag.....321
- A <symbol> can now contain a <local> flag.....325

Changes from February 28, 2023 to December 15, 2023 (from Revision H (February 2023) to Revision I (December 2023)) **Page**

- Added --auto_stream and --assume_addresses_ok_for_stream options.....24
- Added information about advice diagnostics.....44
- Added information about corresponding --opt_level to loop-related optimization descriptions.....56
- Changed "trip count" to "iteration count" throughout to match new software pipelined loop information.....62
- Updated software pipelining example to match new embedded software pipelined loop information.....62
- Added information about using performance advice diagnostics.....69
- Added advice diagnostic to suggest improvement to loop performance.....69
- Added information about corresponding --opt_level to loop-related optimization descriptions.....71
- Added unroll-and-jam optimization.....74
- Removed statement that Streaming Engines and Streaming Address Generators cannot be used automatically.....76
- Added descriptions of --auto_stream and --assume_addresses_ok_for_stream options.....84
- Added information about potential issues with automatic use of auto streaming.....85
- Added information about tuning automatic use of auto streaming.....85
- Added that the --auto_stream, --assume_addresses_ok_for_stream, and --diag_suppress options may be used with the FUNCTION_OPTIONS pragma.....115
- Added note about partial element predication of vectors.....146

Changes from August 5, 2022 to February 28, 2023 (from Revision G (August 2022) to Revision H (February 2023)) **Page**

- Added the --silicon_errata command-line option.....24
- Added the --silicon_errata command-line option.....34
- Documented predefined macros for ptrdiff_t, size_t, wchar_t, and C7000 vector size.....39
- Added description of Streaming Engines.....75
- Added description of Streaming Address Generator.....76
- Described the advantages of using Streaming Engines and Streaming Address Generators.....76
- Documented macros for use with the MMA and Scalable Vector Programming.....151
- Clarified use of linker-defined symbols, including when and how to use the _symval() operator.....280

Changes from October 22, 2021 to August 5, 2022 (from Revision F (October 2021) to Revision G (August 2022)) **Page**

- The --strict_compatibility linker option no longer has any effect and has been removed from the documentation.....30
- Removed documentation of the --rom linker option, which is not supported.....30
- Added the --fp_single_precision_constant compiler option.....33
- Added the --mma_version=2_256 setting for C7504.....34
- Added the --silicon_version=7504 command-line option.....35
- Added a predefined macro name for C7504 devices.....39
- Added links to additional information to the list of --opt_level optimizations.....56
- Added information about corresponding --opt_level to some optimization descriptions.....71

Copyright © 2026 Texas Instruments Incorporated

- Native vector types are now called "TI vector types" and Boolean vector types have been added..... 95
- Revised information related to OpenCL conformance.....95
- Examples have been updated to use constructors to initialize elements of a vector rather than cast/scalar-widening syntax..... 136
- Accessors are provided for swizzle operators. The examples have been updated to reflect this change..... 142
- Added examples for reinterpretation of Boolean vectors.....145
- Added information and examples of Boolean vectors..... 146
- Added section on scalable vector programming.....151
- The --strict_compatibility linker option no longer has any effect and has been removed from the documentation..... 228
- Removed documentation of the --rom linker option, which is not supported.....228
- Documented that the --absolute_exe and --relocatable options may not be used together..... 230
- Added a predefined macro name for C7504 devices..... 233
- Correct the description of the linker's include file search path..... 233

Changes from March 19, 2021 to October 22, 2021 (from Revision E (March 2021) to Revision F (October 2021)) **Page**

- Added the --mma_version command-line option.....24
- Added the --mma_version command-line option.....34
- Added the --silicon_version=7120 command-line option.....35
- Added a predefined macro name for C7120 devices..... 39
- Link-time optimization not supported for files compiled with different --silicon_version or --mma_version..... 61
- Specify that C7120 and later ISAs can be configured to allow predicated SA loads..... 82
- Added several examples for load operations..... 83
- The SET_DATA_SECTION pragma takes precedence over the --gen_data_subsections=on option..... 123
- Document that all C7000 code and data must lie within 2 GB of virtual address space..... 155
- Removed runtime support arithmetic functions that are no longer in the RTS library..... 169
- Clarified information about string handling functions.....179
- Added information about time and clock RTS functions.....179
- The SET_DATA_SECTION pragma takes precedence over the --gen_data_subsections=on option..... 220
- Added a predefined macro name for C7120 devices..... 233
- Document that all C7000 code and data must lie within 2 GB of virtual address space..... 252

Changes from December 15, 2020 to March 19, 2021 (from Revision D (December 2020) to Revision E (March 2021)) **Page**

- Vector predicated stores generated by the compiler may trigger page fault exceptions in certain situations. This issue can be corrected in the linker command file.....297

Changes from February 28, 2020 to December 15, 2020 (from Revision C (February 2020) to Revision D (December 2020)) **Page**

- Updated the numbering format for tables, figures, and cross-references throughout the document..... 13
- Removed references to the Processors wiki throughout the document..... 13
- Clarified that --opt_level=4 must be placed before --run_linker option.....61
- Added predicated loads for Streaming Address Generator and examples that cause well-defined vs. unspecified behavior for store and load operations depending on the SA configuration.....83
- Documented that C11 atomic operations are not supported..... 88
- Updated information about the size of enum types..... 94

Copyright © 2026 Texas Instruments Incorporated

- Clarify interaction between --opt_level and FUNCTION_OPTIONS pragma....115
- Added C++ attribute syntax for attributes that correspond to the MUST_ITERATE pragma....117
- Added C++ attribute syntax for attributes that correspond to the UNROLL pragma....124
- Documented that C11 atomic operations are not supported....128
- Added example using the location attribute....133
- Added information about default for --gen_func_subsections option....220
- Corrected information about default for --gen_data_subsections option....220

The following table lists changes made to this document between v7.4 of the Code Generation Tools (SPRU187) and SPRUI03C (when the document production system was changed). The left column identifies the version of this document in which that particular change appeared.

Earlier Revisions

Version Added	Chapter	Location	Additions / Modifications / Deletions
SPRUIG8C	C/C++ Language	Section 5.3.2, Section 5.14	Added note that vectors cannot be passed via stdarg and cannot be passed to printf().
SPRUIG8C	C/C++ Language	Section 5.8.23, Section 5.8.30, Section 5.8.34, Section 5.13.3	C++ attribute syntax is available to correspond to the MUST_ITERATE, PROB_ITERATE, and UNROLL pragmas.
SPRUIG8C	C/C++ Language	Section 5.8.28	The `#pragma once` is now documented for use in header files.
SPRUIG8C	Run-Time Environment, Linker	Section 6.7.2.1, Section 12.4.33	Clarified that zero initialization takes place only if the --rom_model linker option is used, not if the --ram_model option is used.
SPRUIG8C	Program Loading	Section 9.3.2.3	Corrected information about RAM and ROM model use of CINIT.
SPRUIG8C	Linking, Linker	Section 11.3.4, Section 12.4.24	Clarified that either --rom_model or --ram_model is required if only the linker is being run, but --rom_model is the default if the compiler runs on C/C++ files on the same command line.
SPRUIG8C	Linker	Section 12.5.4.2, Section 12.5.9.7, and Section 12.5.9.8	Added LAST operator to define a symbol with the run-time address of the last allocated byte in the related memory range.
SPRUIG8B	Optimizing	Section 4.15	Updated types for Streaming Engine and Streaming Address Generator.
SPRUIG8B	Language	Section 5.2	MISRA C 2004 checking is not supported at this time.
SPRUIG8B	Language	Section 5.14.6	Conversion modifiers for vector data types are not supported at this time.
SPRUIG8A	Introduction, Language	Section 1.3, Section 5.1, and Section 5.2	Support for C11 and C++14 has been added.
SPRUIG8A	Optimization	Section 4.15	Information about the vector length and element size for the Steaming Engine and Streaming Address Generator has been added.

Copyright © 2026 Texas Instruments Incorporated

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Copyright © 2026, Texas Instruments Incorporated

Last updated 10/2025