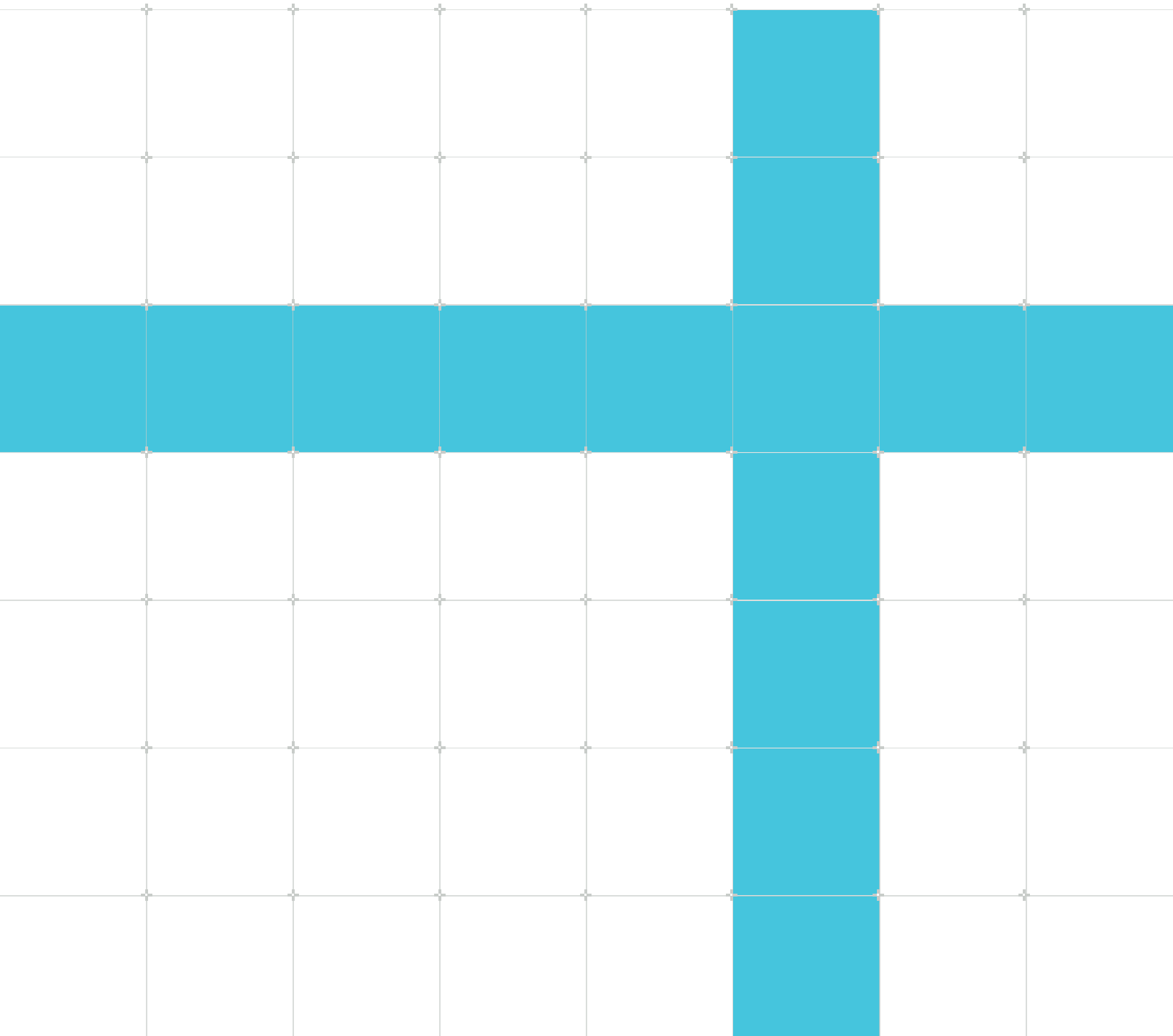
Arm® Compiler for Embedded FuSa

Version 6.22.2 LTS

User Guide

Non-Confidential

Issue 01
109442_6.22.2LTS_01_en



Arm® Compiler for Embedded FuSa

User Guide

Release information

Document history

Issue	Date	Confidentiality	Change
062202LTS-01	28 August 2025	Non-Confidential	Arm Compiler for Embedded FuSa 6.22.2 LTS Release.
062201LTS-01	21 August 2024	Non-Confidential	Arm Compiler for Embedded FuSa 6.22.1 LTS Release.

Proprietary Notice

This document may include technical inaccuracies or typographical errors. THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

List of Figures.....13

List of Tables..... 15

1. Introduction....17
1.1 Conventions....17
1.2 Useful resources....18
1.3 Other information....20

2. Getting Started....21
2.1 Tools and libraries provided with Arm Compiler for Embedded FuSa 6....21
2.2 Application development....23
2.3 About the Arm Compiler for Embedded FuSa toolchain assemblers....25
2.4 System requirements and installation....26
2.5 Accessing Arm Compiler for Embedded FuSa from Arm Development Studio....29
2.6 Accessing Arm Compiler for Embedded FuSa from the Arm Keil MDK....30
2.7 Compiling a Hello World example....30
2.8 Using the integrated assembler....33
2.9 Running bare-metal images....36
2.10 Architectures supported by Arm Compiler for Embedded FuSa 6....37
2.11 Using Arm Compiler for Embedded FuSa securely in a shared environment....38
2.12 Providing source code to Arm support....39
2.13 Build attributes....39

3. Using Common Compiler Options....42
3.1 Mandatory armclang options....42
3.2 Common Arm Compiler for Embedded FuSa toolchain options....44
3.3 Selecting source language options....47
3.4 Selecting optimization options....52
3.5 Building to aid debugging....57
3.6 Linking object files to produce an executable....58
3.7 Linker options for mapping code and data to target memory....59
3.8 Passing options from the compiler to the linker....60
3.9 Controlling diagnostic messages....61
3.10 Selecting floating-point options....68
3.11 Compilation tools command-line option rules....71

4. Writing Optimized Code....73
4.1 Effect of the volatile keyword on compiler optimization....73
4.2 Optimizing loops....77
4.3 Inlining functions....84

4.4 Stack use in C and C++....87
4.5 Packing data structures....90
4.6 Optimizing for code size or performance....96
4.7 Methods of minimizing function parameter passing overhead....98
4.8 Optimizing across modules with Link-Time Optimization....98
4.8.1 Enabling Link-Time Optimization....100
4.8.2 Restrictions with Link-Time Optimization....100
4.8.3 Link-Time Optimization examples....102
4.8.4 Removing unused code across multiple object files....103
4.9 Scatter file section or object placement with Link-Time Optimization....105
4.10 How optimization affects the debug experience....112
4.11 Literal pool options in armclang....113

5. Assembling Assembly Code....114
5.1 Assembling GNU syntax and armasm assembly code....114
5.2 How to get a backtrace through assembler functions....116
5.3 Preprocessing assembly code....117

6. Using Assembly and Intrinsics in C or C++ Code....119
6.1 Using intrinsics....119
6.2 Custom Datapath Extension support....122
6.3 Writing inline assembly code....123
6.4 Calling assembly functions from C and C++....127

7. SVE Coding Considerations with Arm Compiler for Embedded FuSa 6....129
7.1 Assembling SVE code....129
7.2 Disassembling SVE object files....131
7.3 Running a binary in an AEMv8-A Base Fixed Virtual Platform (FVP)....132
7.4 Embedding SVE assembly code directly into C and C++ code....136
7.5 Using SVE and SVE2 intrinsics directly in your C code....141

8. Alignment support in Arm Compiler for Embedded FuSa 6....149
8.1 Aligned and unaligned accesses....151
8.2 Unaligned access support in Arm Compiler for Embedded FuSa....154
8.3 Alignment at the source code and compilation level....157
8.4 Example of padding between structure elements....158
8.5 Alignment and unsafe casting....162

8.6 Example of casting a char pointer to an int pointer....162
8.7 Instruction alignment of functions and loops....165
8.8 Alignment and linking....166

9. Building for different target architectures....168
9.1 How to build for an Armv8-R AArch64 target without hardware floating-point support....168

10. Mapping Code and Data to the Target....170
10.1 What the linker does to create an image....170
10.1.1 What you can control with a scatter file....171
10.1.2 Interaction of OVERLAY and PROTECTED attributes with armlink merge options....171
10.2 Support for position independent code....172
10.3 Placing data items for target peripherals with a scatter file....179
10.4 Placing the stack and heap with a scatter file....181
10.5 Root region....182
10.5.1 Effect of the ABSOLUTE attribute on a root region....182
10.5.2 Effect of the FIXED attribute on a root region....183
10.6 Placing functions and data in a named section....185
10.7 Loading armlink-generated ELF files that have complex scatter-files....188
10.8 Placement of functions and data at specific addresses....190
10.8.1 Placement of __at sections at a specific address....191
10.8.2 Restrictions on placing __at sections....192
10.8.3 Automatic placement of __at sections....192
10.8.4 Manual placement of __at sections....193
10.8.5 Place a key in flash memory with an __at section....194
10.8.6 Placing constants at fixed locations....195
10.8.7 Placing jump tables in ROM....196
10.8.8 Placing a variable at a specific address without scatter-loading....198
10.8.9 Placing a variable at a specific address with scatter-loading....199
10.9 Bare-metal Position Independent Executables....200
10.10 Placement of Arm C and C++ library code....203
10.10.1 Placement of code in a root region....204
10.10.2 Placement of Arm C library code....204
10.10.3 Placing Arm C++ library code....205
10.11 Manual placement of unassigned sections....206
10.11.1 Default rules for placing unassigned sections....207
10.11.2 Command-line options for controlling the placement of unassigned sections....208

10.11.3 Prioritizing the placement of unassigned sections....208
10.11.4 Specify the maximum region size permitted for placing unassigned sections.... 209
10.11.5 Examples of using placement algorithms for .ANY sections.... 210
10.11.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority.... 212
10.11.7 Examples of using sorting algorithms for .ANY sections....213
10.11.8 Behavior when .ANY sections overflow because of linker-generated content.... 215
10.12 Placing veneers with a scatter file....219
10.13 Preprocessing a scatter file....220
10.14 Reserving an empty block of memory....221
10.14.1 Characteristics of a reserved empty block of memory....221
10.14.2 Example of reserving an empty block of memory....222
10.15 Alignment of regions to page boundaries....223
10.16 Alignment of execution regions and input sections....224

11. Overlay support in Arm Compiler for Embedded FuSa 6....226
11.1 Automatic overlay support....226
11.1.1 Automatically placing code sections in overlay regions....227
11.1.2 Overlay veneer.... 229
11.1.3 Overlay data tables....229
11.1.4 Limitations of automatic overlay support....230
11.1.5 About writing an overlay manager for automatically placed overlays....231
11.2 Manual overlay support.... 232
11.2.1 Manually placing code sections in overlay regions.... 233
11.2.2 Writing an overlay manager for manually placed overlays....235

12. Embedded Software Development.... 241
12.1 Default compilation tool behavior....241
12.2 C library structure.... 242
12.3 Default memory map....243
12.4 Application startup.... 244
12.5 Tailoring the C library to your target hardware....246
12.6 Reimplement the C library functions....247
12.7 Tailoring the image memory map to your target hardware.... 249
12.8 About the scatter-loading description syntax....250
12.9 Root regions....251
12.10 Region Table format.... 251
12.11 Placing the stack and heap....253

12.12 Run-time memory models....254
12.13 Reset and initialization....255
12.14 The vector table....257
12.14.1 Vector table for AArch32 A and R profiles....257
12.14.2 Vector table for M-profile architectures....258
12.14.3 Vector Table Offset Register....259
12.15 ROM and RAM remapping....259
12.16 About Run-Time Type Information....260
12.17 Avoid linking in the Arm Compiler for Embedded FuSa libraries....261
12.17.1 Avoid linking in the Arm C library....264
12.17.2 Avoid linking in the Arm C++ libraries....266
12.17.3 Avoid linking in Run-Time Type Information....267
12.17.4 C++ functions you can re-implement....269
12.18 Local memory setup considerations....270
12.19 Stack pointer initialization....271
12.20 Hardware initialization....272
12.21 Execution mode considerations....273
12.22 Target hardware and the memory map....273
12.23 Execute-only memory....274
12.24 Building applications for execute-only memory....275
12.25 Compiling with -mexecute-only generates an empty .text section....276
12.26 Integer division by zero errors in C and C++ code....279
12.27 Floating-point division by zero errors in C and C++ code....280
12.28 Dealing with leftover debug data for code and data removed by armlink....282
12.29 Building images that are compatible with third-party tools....283

13. Security features supported in Arm Compiler for Embedded FuSa....285
13.1 How optimization can interfere with security....291
13.2 Hardware errata and vulnerabilities....292
13.3 Overview of building Secure and Non-secure images with the Armv8-M Security Extension....293
13.4 Building a Secure image using the Armv8-M Security Extension....297
13.5 Building a Non-secure image that can call a Secure image....301
13.6 Building a Secure image using a previously generated import library....302
13.7 Armv8.1-M PACBTI extension mitigations against ROP and JOP style attacks....306
13.8 Overview of the Realm Management Extension....311
13.9 Overview of memory tagging....311

13.10 Overview of Control Flow Integrity... 313
13.11 Overview of Undefined Behavior Sanitizer... 315
13.12 Overview of Straight-Line Speculation hardening... 316
13.13 Memory-safety best practices... 317

14. Thread-Local Storage... 319
14.1 AArch64 TLS local-exec static linking example... 320
14.2 Build and clean scripts for the AArch64 TLS local-exec static linking example... 324
14.3 Run scripts for the AArch64 TLS local-exec static linking example... 326
14.4 Scatter file for the AArch64 TLS local-exec static linking example... 326
14.5 Assembly source files for the AArch64 TLS local-exec static linking example... 328
14.6 C source files for the AArch64 TLS local-exec static linking example... 350

15. Overview of the Linker... 386
15.1 armlink command-line syntax... 387
15.2 What the linker does when constructing an executable image... 387
15.3 What the linker can accept as input... 388
15.4 What the linker outputs... 388

16. Getting Image Details... 390
16.1 Identifying the source of some link errors... 390
16.2 Example of using the --info linker option... 391
16.3 How to find where a symbol is placed when linking... 394

17. SysV Dynamic Linking... 396
17.1 Build a SysV shared object... 396
17.2 Build a SysV executable... 397

18. Overview of the fromelf Image Converter... 399
18.1 fromelf execution modes... 400
18.2 Getting help on the fromelf command... 400
18.3 fromelf command-line syntax... 400

19. Using fromelf... 402
19.1 General considerations when using fromelf... 402
19.2 Examples of processing ELF files in an archive... 402
19.3 Options to protect code in image files with fromelf... 403
19.4 Options to protect code in object files with fromelf... 404

19.5 Option to print specific details of ELF files... 406
19.6 Using fromelf to find where a symbol is placed in an executable ELF image... 406

20. Overview of the Arm Librarian... 409
20.1 Considerations when working with library files... 409
20.2 armar command-line syntax... 410
20.3 Option to get help on the armar command... 410

21. Overview of the armasm Legacy Assembler... 411
21.1 How the assembler works... 412

22. Supporting reference information... 414
22.1 Support level definitions... 414
22.2 Standards compliance in Arm Compiler for Embedded FuSa 6... 419
22.3 Compliance with the ABI for the Arm Architecture (Base Standard)... 420
22.4 GCC compatibility provided by Arm Compiler for Embedded FuSa 6... 422
22.5 Locale support in Arm Compiler for Embedded FuSa 6... 422
22.6 Toolchain environment variables... 422
22.7 Clang and LLVM documentation... 424
22.8 Extensions that are considered qualified features within the scope of functional safety certification... 425
22.9 Extensions that are outside the scope of functional safety certification... 426
22.10 typinfo.s example source code... 428
22.11 Further reading... 433

A. Arm Compiler for Embedded FuSa User Guide Changes... 435
A.1 Changes for the Arm Compiler for Embedded FuSa User Guide... 435

List of Figures

Figure 2-1: A typical tool usage flow diagram....24
Figure 4-1: Structure without packing attribute or pragma....92
Figure 4-2: Structure with attribute packed....92
Figure 4-3: Structure with pragma pack with 1 byte alignment....93
Figure 4-4: Structure with pragma pack with 2 byte alignment....93
Figure 4-5: Structure with pragma pack with 4 byte alignment....94
Figure 4-6: Structure with attribute packed on individual member....95
Figure 4-7: Link-Time Optimization....99
Figure 10-1: Position Independent Code layout....174
Figure 10-2: Position Independent Code relative relocations....175
Figure 10-3: Bare-metal PIE....176
Figure 10-4: ROPI and RWPI....178
Figure 10-5: Memory map for fixed execution regions....183
Figure 10-6: .ANY contingency....216
Figure 10-7: Reserving a region for the stack....223
Figure 12-1: C library structure....242
Figure 12-2: Default memory map....243
Figure 12-3: Linker placement rules....244
Figure 12-4: Default initialization sequence....245
Figure 12-5: Retargeting the C library....246
Figure 12-6: Scatter-loading description syntax....250
Figure 12-7: One-region model....254
Figure 12-8: Two-region model....255

Figure 12-9: Initialization sequence..... 256
Figure 22-1: Integration boundaries in Arm Compiler for Embedded 6..... 416

List of Tables

Table 3-1: armclang common options....44
Table 3-2: armlink common options....45
Table 3-3: armar common options....46
Table 3-4: fromelf common options....46
Table 3-5: armasm common options....47
Table 3-6: Supported C and C++ source language variants....48
Table 3-7: Exceptions to the support for the language standards....50
Table 3-9: armclang linker control options....60
Table 3-10: Common diagnostic options....62
Table 3-11: Options for floating-point selection....68
Table 3-12: Floating-point linkage for AArch32....70
Table 4-1: Loop unrolling pragmas....77
Table 4-2: Function inlining....85
Table 4-3: Packing members in a structure or union....91
Table 7-1: Element selection by predicate type svbool_t....143
Table 7-2: Common addressing mode disambiguators....144
Table 8-1: Armv8 AArch32 alignment requirements of load and store instructions....149
Table 8-2: Access alignment for variants of load instructions....152
Table 8-3: Armv8 AArch32 alignment requirements of load and store instructions....152
Table 8-4: Armv7 alignment requirements of load and store instructions....153
Table 10-3: Input section properties for placement of .ANY sections....210
Table 10-4: Input section properties for placement of sections with next_fit....212
Table 10-6: Sort order for descending_size algorithm....214

Table 10-7: Sort order for cmdline algorithm....214
Table 11-1: Using relative offset in overlays....234
Table 12-4: Types of library function....261
Table 13-3: PACRET-M build attributes....308
Table 13-4: Build attributes and linker behavior....310
Table 13-5: --library_security options and linker behavior....310
Table 13-6: Control Flow Integrity schemes supported....314
Table 22-1: Environment variables used by the toolchain....423
Table 22-2: Documented product features and language extensions....425
Table 22-3: Associated open-source Clang/LLVM tests for language extensions....426
Table A-1: Changes between 6.22.1 LTS and 6.22.2 LTS....435
Table A-2: Changes between 6.22 and 6.22.1 LTS....435

1. Introduction

The Arm® Compiler for Embedded FuSa User Guide provides information for users new to Arm Compiler for Embedded FuSa 6.

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

We recommend the following. If you do not follow these recommendations your system might not work.

Your system requires the following. If you do not follow these requirements your system will not work.

You are at risk of causing permanent damage to your system or your equipment, or harming yourself.

This information is important and needs your attention.

A useful tip that might make it easier, better or faster to perform a task.

A reminder of something important that relates to the information you are reading.

1.2 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
Arm Compiler for Embedded FuSa Reference Guide	109443	Non-Confidential
Arm Compiler for Embedded FuSa Migration and Compatibility Guide	109444	Non-Confidential

Arm product resources	Document ID	Confidentiality
Arm Compiler for Embedded FuSa Arm C and C++ Libraries and Floating-Point Support User Guide	109445	Non-Confidential
Arm Compiler for Embedded FuSa Errors and Warnings Reference Guide	109446	Non-Confidential
Arm Support	-	-
Arm Compiler for Linux	-	-
Arm Development Studio Getting Started Guide	101469	Non-Confidential
Arm Development Studio User Guide	101470	Non-Confidential
Arm Compiler for Embedded Licensing Configuration	-	-
Request a license	-	-
Manage Arm Compiler Versions	-	Non-Confidential
User-based licensing User Guide	102516	Non-Confidential
CMSIS 5	-	Non-Confidential

Arm® architecture and specifications	Document ID	Confidentiality
Arm Architecture Reference Manual for A-profile architecture	DDI 0487	Non-Confidential
ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition	DDI 0406	Non-Confidential
A-Profile Architecture	-	Non-Confidential
M-Profile Architecture	-	Non-Confidential
R-Profile Architecture	-	Non-Confidential
ABI for the Arm Architecture	-	Non-Confidential
C Library ABI for the Arm Architecture	-	Non-Confidential
C++ ABI for the Arm Architecture	-	Non-Confidential
C++ Application Binary Interface Standard for the Arm 64-bit Architecture	-	Non-Confidential
DWARF for the Arm Architecture	-	Non-Confidential
ELF for the Arm Architecture	-	Non-Confidential
Exception Handling ABI for the Arm Architecture	-	Non-Confidential
Procedure Call Standard for the Arm Architecture	-	Non-Confidential
Run-time ABI for the Arm Architecture	-	Non-Confidential
Support for Debugging Overlaid Programs	-	Non-Confidential
Addenda to, and Errata in, the ABI for the Arm Architecture	-	Non-Confidential
Whitepaper - Armv8-M Architecture Technical Overview	-	Non-Confidential
Armv8-M Stack Sealing vulnerability	-	Non-Confidential

Non-Arm resources	Document ID	Organization
GCC	-	https://gcc.gnu.org/onlinedocs/gcc
GNU Binutils	-	https://sourceware.org/binutils
Itanium C++ ABI	-	https://itanium-cxx-abi.github.io/cxx-abi
The Security Implications Of Compiler Optimizations On Cryptography - A Review	-	https://arxiv.org
Using Clang as a Compiler	-	https://releases.llvm.org/18.1.0/tools/clang/docs
Automatic variable initialization	-	https://reviews.llvm.org
C++ implementation status in LLVM Clang	-	https://releases.llvm.org/18.1.0/tools/clang/docs
Undefined Behavior Sanitizer	-	https://releases.llvm.org/18.1.0/tools/clang/docs
Update for Universal C Runtime in Windows	-	https://support.microsoft.com

1.3 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2. Getting Started

Arm® Compiler for Embedded FuSa 6 is the most advanced C and C++ compilation toolchain from Arm for Arm® Cortex® and Arm® Neoverse® processors. Arm Compiler for Embedded FuSa 6 is developed alongside the Arm architecture. Therefore, Arm Compiler for Embedded FuSa 6 is tuned to generate highly efficient code for embedded bare-metal applications ranging from small sensors to 64-bit devices.

Arm Compiler for Embedded FuSa 6 is a component of Arm Development Studio and Arm Keil MDK. The features and processors that Arm Compiler for Embedded FuSa 6 supports depend on the product edition. See Compare Editions for Arm Development Studio.

You can use Arm Compiler for Embedded FuSa 6 from Arm Development Studio, Arm Keil MDK, or as a standalone product.

2.1 Tools and libraries provided with Arm Compiler for Embedded FuSa 6

Arm® Compiler for Embedded FuSa 6 combines the optimized tools and libraries from Arm with a modern LLVM-based compiler framework.

Tools available in Arm Compiler for Embedded FuSa 6

The tool components in Arm Compiler for Embedded FuSa 6 are:

`armclang`

The compiler and integrated assembler that compiles C, C++, and GNU-style assembly language sources.

The compiler is based on LLVM and Clang technology. Clang is a compiler front end for LLVM that supports the C and C++ programming languages.

`armasm`

The legacy assembler. Only use `armasm` for legacy Arm-syntax assembly code. Use the `armclang` integrated assembler and GNU syntax for all new assembly files.


Note

The `armasm` legacy assembler is deprecated, and it has not been updated since Arm Compiler 6.10. As a reminder, `armasm` always reports the deprecation warning `A1950W`. To suppress this message, specify the `--diag_suppress=1950` option.

`armasm` does not support:

- Armv8.4-A and later architectures.
- Armv8-R AArch64 targets.
- Certain backported options in Armv8.2-A and Armv8.3-A.

- Assembling Scalable Matrix Extension (SME) instructions.
- Assembling Scalable Vector Extension (SVE) instructions.
- Assembling Armv8.1-M or later architectures, M-profile Vector Extension (MVE).

---

armlink

The linker combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program.

armar

The archiver enables sets of ELF object files to be collected together and maintained in archives or libraries. If you do not change the files often, these collections reduce compilation time as you do not have to recompile from source every time you use them. You can pass such a library or archive to the linker in place of several ELF files. You can also use the archive for distribution to a third-party application developer as you can share the archive without giving away the source code.

fromelf

The image conversion utility can convert Arm ELF images to binary formats. It can also generate textual information about the input image, such as its disassembly, code size, and data size.

C and C++ language and library support in Arm Compiler for Embedded FuSa 6

`armclang` inherits the C and C++ language from `clang`. Therefore, Arm progressively updates the support level based on `clang`. However, there might be a mismatch between the C and C++ library support and the language support. For example, some library features might not apply to embedded development, such as `filesystem` in the C++ library.

For more information, see Selecting source language options.

Arm C++ libraries

The Arm C++ libraries are based on the LLVM libc++ project:

- The libc++abi library is a runtime library providing implementations of low-level language features.
- The libc++ library provides an implementation of the ISO C++ library standard. It depends on the functions that are provided by libc++abi.

We do not guarantee the compatibility of C++ compilation units compiled with different major or minor versions of Arm Compiler for Embedded FuSa and linked into a single image. Therefore, we recommend that you always build your C++ code from source with a single version of the toolchain.

You can mix C++ with C code or C libraries.

Arm C libraries

The Arm C libraries provide:

- An implementation of the library features as defined in the C standards.
- Nonstandard extensions common to many C libraries.
- POSIX extended functionality.
- Functions standardized by POSIX.


Note

Comments inside source files and header files that are provided by Arm might not be accurate and must not be treated as documentation about the product.

For C and C++ language support and libc++ library support in Arm Compiler for Embedded FuSa 6, see:

- C language
- C++ language

Also, see *List of known unsupported features* in Support level definitions.

Related information

Compiling a Hello World example on page 30
Common Arm Compiler for Embedded FuSa toolchain options on page 44
-S (armclang)
Arm C and C++ library directory structure
C++ implementation status in LLVM Clang

2.2 Application development

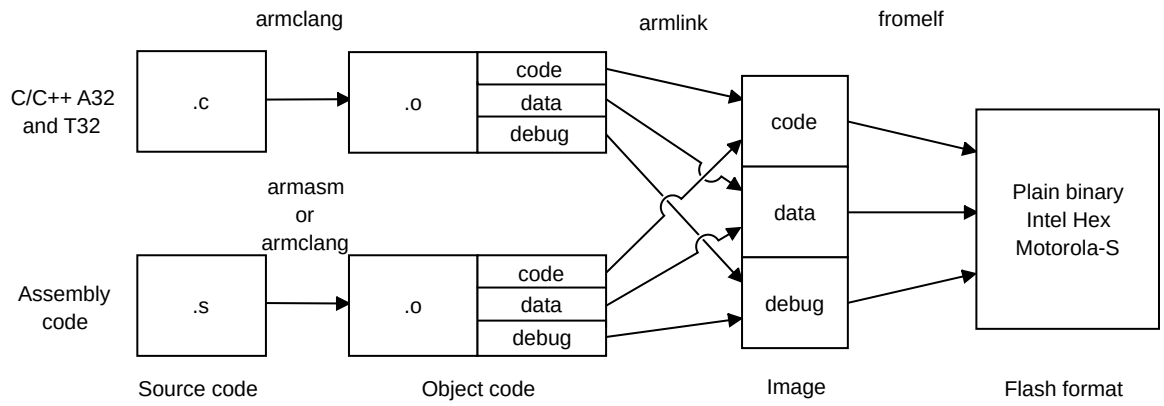
A typical application development flow might involve working with multiple tools.

The development flow might involve:

- Developing C/C++ source code for the main application (`armclang`).
- Developing assembly source code for near-hardware components, such as interrupt service routines (`armclang`, or `armasm` for legacy assembly code).
- Linking all objects together to generate an image (`armlink`).
- Converting an image to flash format in plain binary, Intel Hex, and Motorola-S formats (`fromelf`).

The following figure shows how the compilation tools are used for the development of a typical application:

Figure 2-1: A typical tool usage flow diagram



Arm® Compiler for Embedded FuSa 6 has more functionality than the set of product features that is described in the documentation. The various features in Arm Compiler for Embedded FuSa 6 can have different levels of support and guarantees. For more information, see Support level definitions.

Note

- If you are migrating your toolchain from Arm Compiler 5 to Arm Compiler for Embedded FuSa 6, see the Migration and Compatibility Guide. It contains information on how to migrate your source code and toolchain build options.
- For a list of Arm Compiler for Embedded FuSa 6 documents, see the Arm Compiler for Embedded documentation index on Arm Developer.

Note

Be aware of the following:

- Generated code might be different between two Arm Compiler for Embedded FuSa releases.
- For a feature release, there might be significant code generation differences.

Related information

Compiling a Hello World example on page 30
Common Arm Compiler for Embedded FuSa toolchain options on page 44
-S (armclang)

2.3 About the Arm Compiler for Embedded FuSa toolchain assemblers

The Arm® Compiler for Embedded FuSa toolchain provides different assemblers.

They are:

- The `armclang` integrated assembler. Use this to assemble assembly language code written in GNU syntax.
- An optimizing inline assembler built into `armclang`. Use this to assemble assembly language code written in GNU syntax that is used inline in C or C++ source code.
- The freestanding legacy assembler, `armasm`. Use `armasm` to assemble existing A64, A32, and T32 assembly language code written in armasm syntax.


Note

The `armasm` legacy assembler is deprecated, and it has not been updated since Arm Compiler 6.10. As a reminder, `armasm` always reports the deprecation warning `A1950W`. To suppress this message, specify the `--diag_suppress=1950` option.

`armasm` does not support:

- Armv8.4-A and later architectures.
- Armv8-R AArch64 targets.
- Certain backported options in Armv8.2-A and Armv8.3-A.
- Assembling Scalable Matrix Extension (SME) instructions.
- Assembling Scalable Vector Extension (SVE) instructions.
- Assembling Armv8.1-M or later architectures, M-profile Vector Extension (MVE).


Note

The command-line option descriptions and related information in the *Arm Compiler for Embedded FuSa Reference Guide* describe all the features that Arm Compiler for Embedded FuSa supports. Any features not documented are not supported and are used at your own risk. You are responsible for making sure that any generated code using community features is operating correctly. See Support level definitions.

Related information

Using Assembly and Intrinsics in C or C++ Code on page 119
Assembling GNU syntax and armasm assembly code on page 114
Arm Compiler for Embedded FuSa Reference Guide

2.4 System requirements and installation

The system requirements for running Arm® Compiler for Embedded FuSa and instructions to guide you through the installation process.

System Requirements

Arm Compiler for Embedded FuSa 6 is available for the following:

- x86_64 Windows
- x86_64 Windows for Arm® Keil® Microprocessor Development Kit (MDK)
- x86_64 Linux
- AArch64 Linux

For more information on system requirements, see the Release Notes that are available at Release Notes for Arm Compiler for Embedded FuSa 6.22.2.

Installing Arm Compiler for Embedded FuSa

You can install Arm Compiler for Embedded FuSa as a standalone product on supported Windows and Linux platforms. If you use Arm Compiler for Embedded FuSa as part of a development suite such as Arm Development Studio or Arm Keil MDK, installing the development suite also installs Arm Compiler for Embedded FuSa. The following instructions are for installing Arm Compiler for Embedded FuSa as a standalone product.

The Linux installers of Arm Compiler for Embedded FuSa might be vulnerable to the CVE-2022-43701 permission-based attack. For more information, see Installer vulnerabilities CVE-2022-43701, CVE-2022-43702, and CVE-2022-43703.

Prerequisites

1. Click the link in the **Product Download Hub page** column of the Arm Compiler downloads index to download the installer for your version. The download pack provided for use with Keil MDK is not suitable for standalone use.
2. Obtain a user-based license. Contact your Arm sales representative or Request a license. For more information about user-based licensing, see the User-based licensing overview.

Installing a standalone Arm Compiler for Embedded FuSa on x86_64 Windows platforms

To install Arm Compiler for Embedded FuSa as a standalone product on Windows for x86_64, you need the `Arm Compiler for Embedded FuSa 6.22.<u>.msi` installer on your machine, where `6.22.<u>` is the product version number:

1. Open a command prompt with administrative privileges.
2. Unzip the `ARMCompiler6.22,<u>_standalone_win-x86_64.zip`.
3. Run `win-x86_64\Arm Compiler for Embedded FuSa 6.22.<u>.msi`.
4. Follow the on-screen installation instructions.

5. Make sure you activate your user-based license and that your user-based license server is running. For more information about User-based licensing, see the User-based Licensing User Guide.

If you have an older version of Arm Compiler for Embedded FuSa 6 and you want to upgrade, we recommend that you uninstall the older version of Arm Compiler for Embedded FuSa 6 before installing the new version of Arm Compiler for Embedded FuSa 6.

Arm Compiler for Embedded FuSa requires the Universal C Runtime in Windows to be installed. For more information, see Update for Universal C Runtime in Windows.

Installing a standalone Arm Compiler for Embedded FuSa on x86_64 Windows platforms and accepting the EULA

You can install Arm Compiler for Embedded FuSa as a standalone product on Windows for x86_64 using the Windows installer tool `msiexec.exe`. This tool allows you to automatically accept the Arm EULA as part of the installation with the Arm-specific option `EULA=1`.

To install:

1. Open a command prompt with administrative privileges.
2. Unzip the file `ARMCompiler6.22.<u>_standalone_win-x86_64.zip`.
3. You must read the EULA before accepting it on the command-line. The EULA is available in the file `license_terms\license_agreement.txt`.
4. Run `msiexec.exe` to install to the default location:

```
msiexec.exe /i "win-x86_64\\Arm Compiler for Embedded FuSa 6.22.<u>.msi" /qn /lv
 "<path to log file>" EULA=1
```

   To install to a different location, add the `INSTALLDIR` option, for example:

```
msiexec.exe /i "win-x86_64\\Arm Compiler for Embedded FuSa 6.22.<u>.msi" /qn /lv
 "<path to log file>" EULA=1 INSTALLDIR=<path to install dir>
```

   For more information about the `msiexec.exe` options, see msiexec.

Installing a standalone Arm Compiler for Embedded FuSa on x86_64 Linux platforms

To install Arm Compiler for Embedded FuSa as a standalone product on x86_64 Linux platforms, you need the `install_x86_64.sh` installer on your machine:

1. Run `install_x86_64.sh` normally, without using the `source` Linux command.
2. Follow the on-screen installation instructions.
3. Make sure you activate your user-based license and that your user-based license server is running. For more information about User-based licensing, see the User-based Licensing User Guide.

To allow you to execute the `armclang` binary, it is dynamically linked to a copy of libstdc++ that is installed under your chosen directory as part of Arm Compiler for Embedded FuSa. libstdc++ is not the C++ standard library that you use to build a C++ project for Arm target devices.

Installing a standalone Arm Compiler for Embedded FuSa on AArch64 Linux platforms

To install Arm Compiler for Embedded FuSa as a standalone product on AArch64 Linux platforms, you need the `install_aarch64.sh` installer on your machine:

1. Run `install_aarch64.sh` normally, without using the `source` Linux command.
2. Follow the on-screen installation instructions.
3. Make sure you activate your user-based license and that your user-based license server is running. For more information about User-based licensing, see the User-based Licensing User Guide.

To allow you to execute the `armclang` binary, it is dynamically linked to a copy of libstdc++ that is installed under your chosen directory as part of Arm Compiler for Embedded FuSa. libstdc++ is not the C++ standard library that you use to build a C++ project for Arm target devices.

Using the checksums.txt file to verify the installation

Arm Compiler for Embedded FuSa includes the checksum file `<install_directory>\sw\checksums.txt`. This file contains checksums of the files in the Arm Compiler for Embedded FuSa installation. The checksums are calculated using the SHA256 hash algorithm.

Linux installation

To verify the installed files on Linux run the following command in `<install_dir>`:

```
shasum -c ./sw/checksums.txt
```

Windows installation

1. To verify the installed files on Windows create the file `check_checksum.bat` containing the following commands:

```
@ECHO OFF
setlocal enabledelayedexpansion
REM Cycle through each line of checksums.txt
for /F "tokens=*" %%L in (sw\checksums.txt) do (
  REM For each line grab two tokens: %%a (hash) %%b (file path)
  for /F "tokens=1,2 delims= " %%a in ("%%L") do (
    REM Run certutil on the file path, and cycle over its output line
    for /F "usebackq tokens=* skip=1" %%C in (`certutil -hashfile "%%b"
 SHA256`) do (
      REM We only need the 2nd of 3 lines output by certutil
      REM (skip=1 ignores the first)
      set var=""
      REM Searching for the string 'CertUtil' allows us to ignore the 3rd
      for /F "usebackq delims=" %%x in (`echo "%%C"^|findstr /v "CertUtil"`)
 do (
        set var=%%x
      )
      REM If this is the 2nd 'hash' line of certutil, then it is time to
 compare
      if not "!var!" == """" (call :compare_hashes %%b %%a !var!)
    )
  )
)
```

```
echo All hashes match.
EXIT /B 0

:compare_hashes
echo Checking file: %1
echo ... Expected checksum: "%2"
echo ... Received checksum: %3
if %3 == "%2" (echo ... Success) else (echo ... Failure && EXIT /B 11)
```

2. Run `check_checksum.bat` in `<install_dir>`.

Uninstalling a standalone Arm Compiler for Embedded FuSa

To uninstall Arm Compiler for Embedded FuSa on Windows, use the Control Panel:

1. Select **Control Panel > Programs > Programs and Features > Uninstall a program**.
2. Select the version that you want to uninstall, for example **Arm Compiler for Embedded FuSa 6.22.2**.
3. Click **Uninstall**.

To uninstall Arm Compiler for Embedded FuSa on Linux, delete the Arm Compiler for Embedded FuSa installation directory for the compiler version you want to delete.

For more information on installation, see the Release Notes that are available at Release Notes for Arm Compiler for Embedded FuSa 6.22.2.

Related information

Accessing Arm Compiler for Embedded FuSa from Arm Development Studio on page 29
Accessing Arm Compiler for Embedded FuSa from the Arm Keil MDK on page 29

2.5 Accessing Arm Compiler for Embedded FuSa from Arm Development Studio

Arm® Development Studio is a development suite that provides Arm Compiler for Embedded FuSa as a built-in toolchain.

For more information, see Create a new C or C++ project in the *Arm Development Studio Getting Started Guide*.

Related information

System requirements and installation on page 25

2.6 Accessing Arm Compiler for Embedded FuSa from the Arm Keil MDK

Arm® Keil® MDK is a microprocessor development suite that provides the µVision® IDE, and Arm Compiler for Embedded as a built-in toolchain.

To download Arm Compiler for Embedded FuSa:

- For MDK version 5, see Manage Arm Compiler Versions.
- For MDK version 6, see Installing other tools.

Related information

System requirements and installation on page 25

2.7 Compiling a Hello World example

These examples show how to use the Arm® Compiler for Embedded FuSa toolchain to build and inspect an executable image from C/C++ source files.

A simple example

The source code that is used in the examples is a single C source file, `hello.c`, to display a greeting message:

```
#include <stdio.h>

int main() {
  printf("Hello World\n");
  return 0;
}
```

Building an executable in a single step

For simple programs, you can use a single command to compile the source code file to an executable image.

You must first decide which target the executable is to run on. An Armv8-A target can run in different states:

- AArch64 state targets execute A64 instructions using 64-bit and 32-bit general-purpose registers.
- AArch32 state targets execute A32 or T32 instructions using 32-bit general-purpose registers.

The `--target` option determines which target state to compile for. This option is a mandatory option.

Compiling for an AArch64 target

To create an executable for an AArch64 target in a single step:

```
armclang --target=aarch64-arm-none-eabi hello.c
```

This command creates an executable file with the default name `a.out`. You can use the `-o` option to specify a different name for the executable file.

This example compiles for an AArch64 state target. Because only `--target` is specified, the compiler defaults to generating code that runs on any Armv8-A target. You can also use `-mcpu` to target a specific processor.

Compiling for an AArch32 target

To create an executable for an AArch32 target in a single step:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a53 hello.c
```

There is no default target for AArch32 state. You must specify either `-march` to target an architecture or `-mcpu` to target a processor.

This example uses `-mcpu` to target the Cortex®-A53 processor. The compiler generates code that is optimized specifically for the Cortex-A53, but might not run on other processors.

Use `-mcpu=list` or `-march=list` to see all available processor or architecture options.

Beyond the defaults

Compiler options let you specify precisely how the compiler behaves when generating code.

The Arm Compiler for Embedded FuSa Reference Guide describes all the supported options. Some of the most common options are listed in Common Arm Compiler for Embedded FuSa toolchain options.

Examining the executable

The `fromelf` tool lets you examine a compiled binary, extract information about it, or convert it.

For example, you can:

- Disassemble the code that is contained in the executable:

```
fromelf --text -c a.out

  ...
  main
  0x000081a0:    e92d4800    .H-.    PUSH     {r11,lr}
  0x000081a4:    e1a0b00d    ....    MOV      r11,sp
  0x000081a8:    e24dd010    ..M.    SUB      sp,sp,#0x10
  0x000081ac:    e3a00000    ....    MOV      r0,#0
  0x000081b0:    e50b0004    ....    STR      r0,[r11,#-4]
  0x000081b4:    e30a19cc    ....    MOV      r1,#0xa9cc
  ...
```

- Examine the size of code and data in the executable:

```
fromelf --text -z a.out

  Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name
  10436        492        596         16        348       3468   a.out
  10436        492        596         16          0          0   ROM Totals for
 a.out
```

- Convert the ELF executable image to another format, for example a plain binary file:

```
fromelf --bin --output=outfile.bin a.out
```

See fromelf Command-line Options for the options from the `fromelf` tool.

Compiling and linking as separate steps

For simple projects with small numbers of source files, compiling to an executable image in a single step might be the simplest option. You can compile multiple source files into an executable with a command such as the following:

```
armclang --target=aarch64-arm-none-eabi file1.c file2.c -o image.axf
```

This command compiles the two source files `file1.c` and `file2.c` into an executable file for an AArch64 state target. The `-o` option specifies that the filename of the generated executable file is `image.axf`.

However, more complex projects might have a large number of source files. It is not efficient to compile every source file at every compilation, because many of the source files are unlikely to change. To avoid compiling unchanged source files, you can compile and link as separate steps. In this way, you can then use a build system (such as `make`) to compile only those source files that have changed, then link the object code together. The `armclang -c` option tells the compiler to compile to object code and stop before calling the linker:

```
armclang -c --target=aarch64-arm-none-eabi file1.c
armclang -c --target=aarch64-arm-none-eabi file2.c
armlink file1.o file2.o -o image.axf
```

These commands do the following:

- Compile `file1.c` to object code, and save using the default name `file1.o`.
- Compile `file2.c` to object code, and save using the default name `file2.o`.
- Link the object files `file1.o` and `file2.o` to produce an executable that is called `image.axf`.

In future, if you modify `file2.c`, you can rebuild the executable by recompiling only `file2.c` then linking the new `file2.o` with the existing `file1.o` to produce a new executable:

```
armclang -c --target=aarch64-arm-none-eabi file2.c
armlink file1.o file2.o -o image.axf
```

Related information

--target (armclang)
-march (armclang)
-mcpu (armclang)
Summary of armclang command-line options

2.8 Using the integrated assembler

These examples show how to use the `armclang` integrated assembler to build an object from assembly source files, and how to call functions in this object from C/ C++ source files.

The integrated assembler sets a minimum alignment of 4 bytes for a `.text` section. However, if you define your own sections with the integrated assembler, then you must include the `.balign` directive to set the correct alignment. For a section containing T32 instructions, set the alignment to 2 bytes. For a section containing A32 instructions, set the alignment to 4 bytes.

The assembly source code

The assembly example is a single assembly source file, `mystrcopy.s`, containing a function to perform a simple string copy operation:

```
    .section   StringCopy, "ax"
    .balign    4

    .global    mystrcopy
    .type      mystrcopy, "function"
mystrcopy:
    ldrb       r2, [r1], #1
    strb       r2, [r0], #1
    cmp        r2, #0
    bne        mystrcopy
    bx         lr
```

The `.section` directive creates a new section in the object file named `StringCopy`. The characters in the string following the section name are the flags for this section. The `a` flag marks this section as allocatable. The `x` flag marks this section as executable.

The `.balign` directive aligns the subsequent code to a 4-byte boundary. The alignment is required for compliance with the *Procedure Call Standard for the Arm Architecture* (AAPCS).

The `.global` directive marks the symbol `mystrcopy` as a global symbol. This enables the symbol to be referenced by external files.

The `.type` directive sets the type of the symbol `mystrcopy` to `function`. This helps the linker use the proper linkage when the symbol is branched to from A32 or T32 code.

Assembling a source file

When assembling code, you must first decide which target the executable is to run on. The `armclang` option `--target` determines which target state to assemble for. This option is a mandatory option.

To assemble the above source file for an Arm®v8-M Mainline target:

```
armclang --target=arm-arm-none-eabi -c -march=armv8-m.main mystrcopy.s
```

This command creates an object file, `mystrcopy.o`.

In this example, there is no default target for A32 state, so you must specify either `-march` to target an architecture or `-mcpu` to target a processor. This example uses `-march` to target the Armv8-M Mainline architecture. The integrated assembler accepts the same options for `--target`, `-march`, `-mcpu`, and `-mfpu` as the compiler.

Use `-mcpu=list` or `-march=list` to see all available options.


Note

Some update releases and architecture extensions might not be fully supported in this release. Where these are described, the level of support is indicated. See Support level definitions.

Examining the executable

You can use the `fromelf` tool to:

- examine an assembled binary.
- extract information about an assembled binary.
- convert an assembled binary to another format.

For example, you can disassemble the code that is contained in the object file:

```
fromelf --text -c mystrcopy.o

    ...
    ** Section #3 'StringCopy' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
        Size   : 14 bytes (alignment 4)
        Address: 0x00000000

        $t.0
        mystrcopy
            0x00000000:    f8112b01    ...+    LDRB     r2,[r1],#1
            0x00000004:    f8002b01    ...+    STRB     r2,[r0],#1
            0x00000008:    2a00        .*      CMP      r2,#0
            0x0000000a:    d1f9        ..      BNE      mystrcopy ; 0x0
            0x0000000c:    4770        pG      BX       lr
    ...
```

The example shows the disassembly for the section `StringCopy` as created in the source file.

Note

The presence of 16-bit opcodes shows that the code is in the T32 instruction set. T32 is the default in this situtation, because Armv8-M Mainline does not support A32 code. For processors that support A32 and T32 code, you can explicitly mark the code as A32 or T32 by adding the GNU assembly `.arm` or `.thumb` directive, respectively, at the start of the source file.

Calling an assembly function from C/C++ code

It can be useful to write optimized functions in an assembly file and call them from C/C++ code. When doing so, ensure that the assembly function uses registers in compliance with the AAPCS.

The C example is a single C source file `main.c`, containing a call to the `mystrcopy` function to copy a string from one location to another:

```
const char *source = "String to copy.";
char *dest;
extern void mystrcopy(char *dest, const char *source);

int main(void) {
    mystrcopy(dest, source);
    return 0;
}
```

An `extern` function declaration has been added for the `mystrcopy` function. The return type and function parameters must be checked manually.

If you want to call the assembly function from a C++ source file, you must disable C++ name mangling by using `extern "C"` instead of `extern`. For the above example, use:

```
extern "C" void mystrcopy(char *dest, const char *source);
```

Compiling and linking the C source file

To compile the above source file for an Armv8-M Mainline target:

```
armclang --target=arm-arm-none-eabi -c -march=armv8-m.main main.c
```

This command creates an object file, `main.o`.

To link the two object files `main.o` and `mystrcopy.o` and generate an executable image:

```
armlink main.o mystrcopy.o -o image.axf
```

This command creates an executable image file `image.axf`.

Related information

Mandatory armclang options on page 42
Summary of armclang command-line options
Sections

2.9 Running bare-metal images

By default, Arm® Compiler for Embedded FuSa produces bare-metal images. Bare-metal images can run without an operating system. The images can run on a hardware target or on a software application that simulates the target, such as Fast Models or Fixed Virtual Platforms.

The linker creates information to initialize global and static objects (data) and uninitialized global and static objects (`.bss`). Bare-metal images initialize the data by copying and decompressing initialized data and set the `.bss` to zero.

See your Arm Integrated Development Environment (IDE) documentation for more information on configuring and running images:

- For Arm Development Studio, see the Arm Development Studio Getting Started Guide and Arm Development Studio User Guide.
- For Arm® Keil® MDK, see Installation in the *Arm Keil Microcontroller Development Kit (MDK) Getting Started Guide*.

By default, the C library in Arm Compiler for Embedded FuSa uses special functions to access the input and output interfaces on the host computer. These functions implement a feature called semihosting. Semihosting is useful when the input and output on the hardware is not available during the early stages of application development.

When you want your application to use the input and output interfaces on the hardware, you must retarget the required semihosting functions in the C library.

See your Arm IDE documentation for more information on configuring debugger settings:

- For Arm Debugger settings, see Configuring a connection to a bare-metal hardware target in the *Arm Development Studio Getting Started Guide*.
- For information on how to disable semihosting in Arm Keil MDK, see ARM: Application Builds Without Error, But Does Not Run.

Outputting debug messages from your application

The semihosting feature enables your bare-metal application, running on an Arm processor, to use the input and output interface on a host computer. This feature requires the use of a debugger that supports semihosting, for example Arm Debugger, on the host computer.

A bare-metal application that uses semihosting does not use the input and output interface of the development platform. When the input and output interfaces on the development platform are available, you must reimplement the necessary semihosting functions to use them.

For more information, see how to use the libraries in semihosting and nonsemihosting environments.

Related information

Arm Development Studio Getting Started Guide

Arm Development Studio User Guide
Semihosting for AArch32 and AArch64

2.10 Architectures supported by Arm Compiler for Embedded FuSa 6

Arm® Compiler for Embedded FuSa supports a number of different architecture profiles.


Note

Some update releases and architecture extensions might not be fully supported in this release. Where these are described, the level of support is indicated. See Support level definitions.

Arm Compiler for Embedded FuSa supports the following architectures for bare-metal targets:

- Armv9-A up to Armv9.5-A.
- Armv8-A up to Armv8.9-A.
- Armv8-R.
- Armv8-M up to Armv8.1-M.
- Armv7-A.
- Armv7-R.
- Armv7-M.
- Armv6-M.

When compiling code, the compiler must know which architecture to target to take advantage of features specific to that architecture.

To specify a target, you must supply the target execution state (AArch32 or AArch64), together with either a target architecture (for example Armv8-A) or a target processor (for example, the Cortex®-A53 processor).

To specify a target execution state (AArch64 or AArch32) with `armclang`, use the mandatory `--target` command-line option:

```
--target=<arch>-<vendor>-<os>-<abi>
```

Supported targets include:

`aarch64-arm-none-eabi`

Generates A64 instructions for AArch64 state. Implies `-march=armv8-a` unless `-march` or `-mcpu` is specified.

arm-arm-none-eabi

Generates A32 and T32 instructions for AArch32 state. Must be used in conjunction with `-march` (to target an architecture) or `-mcpu` (to target a processor).

To generate generic code that runs on any processor with a particular architecture, use the `-march` option. Use the `-march=list` option to see all supported architectures.

To optimize your code for a particular processor, use the `-mcpu` option. Use the `-mcpu=list` option to see all supported processors.


Note

The `--target`, `-march`, and `-mcpu` options are `armclang` options. For all of the other tools, such as `armlink`, use the `--cpu` option to specify target processors and architectures.

Related information

--target (armclang)
-march (armclang)
-mcpu (armclang)
--cpu (armlink)
Arm Glossary

2.11 Using Arm Compiler for Embedded FuSa securely in a shared environment

Arm® Compiler for Embedded FuSa provides features and language support in common with other toolchains. Misuse of these common features and language support can provide access to arbitrary files, execute system commands, and reveal the contents of environment variables.

If deploying Arm Compiler for Embedded FuSa into environments where security is a concern, then Arm strongly recommends that you do all the following:

- Sandbox the tools to limit their access to only necessary files.
- Remove all non-essential environment variables.
- Prevent execution of other binaries.
- Segregate different users from each other.
- Limit execution time.

2.12 Providing source code to Arm support

When you encounter a problem that requires you to provide source code to Arm support, then you might want to create a minimal example that demonstrates the problem.

Preprocessing your source files with the `armclang` option `-E` might be useful when creating the minimal example as part of a support case. To help the investigation, try to send only the single image, object, source file, or function that is causing the issue, together with the command-line options used.

If your source code contains preprocessor macros, it might be necessary to use the compiler to preprocess the source before sharing it. That is, to take account of files added with `#include`, pass the file through the preprocessor as follows:

```
armclang <options> -E sourcefile.c > PPsourcefile.c
```

Where `<options>` are your normal compile switches, such as `-O2`, `-g`, `-I`, `-D`, but without `-c`.

Related information

Common Arm Compiler for Embedded FuSa toolchain options on page 44
-E (armclang)

2.13 Build attributes

`armclang` or a standalone assembler annotate ELF object files with build attributes. `armlink` uses this data to determine the compatibility of the files that it links.


Note

This topic includes descriptions of [COMMUNITY] features. See Support level definitions.


Note

Arm® Compiler for Embedded FuSa supports build attributes only for AArch32.

Build attributes primarily model two kinds of compatibility:

- The compatibility of binary code with target hardware conforming to a revision of the Arm architecture.
- The procedure-call compatibility between functions conforming to variants of the *ABI for the Arm Architecture*.

Build attributes approximate your intentions for the compatibility of the relocatable file produced by the tool when compiling or assembling code. You express the intentions to the tool as configuration options such as `-mcpu` or `-mno-unaligned-access`.

When compiling C and C++ code, `armclang` is in control of code generation and can guarantee that the object file generated conforms to the intention. When using the assembler, you are in control of code generation. In some cases the assembler can check that the source code conforms to the intentions given on the command-line. For example, if the specified processor does not support a particular instruction, the assembler can give an error message that the instruction is not supported. However, some intentions cannot be easily checked by the assembler.

You can use the `armclang` integrated assembler with options that permit using unaligned data accesses or options that affect the passing of arguments. When using such options, you must ensure that the object file generated conforms to the intentions and purpose of the options:

- Compatibility can be given a mathematically precise definition using sets of demands placed on an execution environment.

  For example, a program is compatible with a processor if, and only if, the set of instructions the program might try to execute is a subset of the instructions implemented by that processor.
- Target-related attributes describe the hardware-related demands a relocatable file places on an execution environment through being included in an executable file for that environment.

  For example, target-related attributes record whether use of the Arm® Thumb® Instruction Set Architecture (ISA) is permitted, and at what architectural revision use is permitted. A pair of values for these attributes describes the set of Thumb instructions that code is permitted to execute and that the target processor must implement.
- Procedure call-related attributes describe features of the ABI contract that the ABI allows to vary. Features such as:
  - Whether floating-point parameters are passed in floating-point registers.
  - The size of `wchar_t`.
  - Whether enumerated values are containerized according to their size.

You can also set intentions by using directives in the assembler source code. You can use the `armclang` [COMMUNITY] option `-mdefault-build-attributes` to add the default build attribute directives to your assembly code. To see how `armclang` encodes the build attributes in the assembly code specify the `-S` option. For example, the `-mno-unaligned-access` sets the `Tag_CPU_unaligned_access` attribute to `0`:

```
armclang --target=arm-arm-none-eabi -march=armv8a -mno-unaligned-access -S -o main.s
 main.c
```

```
    .text
    .syntax unified
    .eabi_attribute   67, "2.09"  @ Tag_conformance
    .eabi_attribute   6, 14   @ Tag_CPU_arch
    .eabi_attribute   7, 65   @ Tag_CPU_arch_profile
    .eabi_attribute   8, 1    @ Tag_ARM_ISA_use
    .eabi_attribute   9, 2    @ Tag_THUMB_ISA_use
    ...
```

```
    .eabi_attribute   34, 0   @ Tag_CPU_unaligned_access
    ...
```

If you have a specific language standard that you are targeting for assembler source code, we recommend that you specify the language standard on the command-line. You must specify the language standard because the assembler does not detect non-conformance between the assembler source code and the stated intentions.

Build attributes are encoded in a binary format. To decode the build attributes, use the `fromelf` option `--decode_build_attributes`. To see a human-readable form, use the `--extract_build_attributes` option.

Related information

Addenda to, and Errata in, the ABI for the Arm Architecture
Summary of armclang command-line options
-mdefault-build-attributes, -mno-default-build-attributes
armclang Integrated Assembler
--decode_build_attributes
--extract_build_attributes

3. Using Common Compiler Options

There are many options that you can use to control how Arm® Compiler for Embedded FuSa generates code for your application. There are mandatory and commonly used optional command-line arguments, such as to control target selection, optimization, and debug view.

3.1 Mandatory armclang options

When using `armclang`, you must specify a target on the command-line. Depending on the target you use, you might also have to specify an architecture or processor.

Specifying a target

To specify a target, use the --target option. The following targets are available:

- To generate A64 instructions for AArch64 state, specify `--target=aarch64-arm-none-eabi`.

  
  Note

  For AArch64, the default architecture is Arm®v8-A.

- To generate A32 and T32 instructions for AArch32 state, specify `--target=arm-arm-none-eabi`. To specify generation of either A32 or T32 instructions, use -marm or -mthumb respectively.

  
  Note

  AArch32 has no defaults. You must always specify an architecture or processor.

Specifying an architecture

To generate code for a specific architecture, use the -march option. The supported architectures vary according to the selected target.

To see a list of all the supported architectures for the selected target, use `-march=list`.

Specifying a processor

To generate code for a specific processor, use the -mcpu option. The supported processors vary according to the selected target.

To see a list of all the supported processors for the selected target, use `-mcpu=list`.

It is also possible to enable or disable optional architecture features, by using the `+[no]feature` notation. For a list of the architecture features that your processor supports, see the processor product documentation. See the *Arm Compiler for Embedded FuSa Reference Guide* for a list of architecture features that Arm Compiler for Embedded FuSa supports.

Use `+<feature>` or `+no<feature>` to explicitly enable or disable an optional architecture feature.


Note

Avoid specifying both the architecture (`-march`) and the processor (`-mcpu`) because specifying both has the potential to cause a conflict. The compiler infers the correct architecture from the processor.

- If you want to run code on one particular processor, specify the processor using `-mcpu`. Performance is optimized, but code is only guaranteed to run on that processor. If you specify a value for `-mcpu`, do not also specify a value for `-march`.
- If you want your code to run on a range of processors from a particular architecture, specify the architecture using `-march`. The code runs on any processor implementation of the target architecture, but performance might be impacted. If you specify a value for `-march`, do not also specify a value for `-mcpu`.

Specifying an optimization level

The default optimization level is `-O0`, which does not apply any optimizations. We recommend that you always specify a suitable optimization level. For more information, see Selecting optimization options in the *Arm Compiler for Embedded FuSa Reference Guide*, and the -O option in the *Arm Compiler for Embedded FuSa Reference Guide*.

Examples

These examples compile and link the input file `helloworld.c`:

- To compile for the Armv8-A architecture in AArch64 state, use:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a helloworld.c
```

- To compile for the Armv8-R architecture in AArch32 state, use:

```
armclang --target=arm-arm-none-eabi -march=armv8-r helloworld.c
```

- To compile for the Armv8-M architecture mainline profile, use:

```
armclang --target=arm-arm-none-eabi -march=armv8-m.main helloworld.c
```

- To compile for a Cortex®-A53 processor in AArch64 state, use:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 helloworld.c
```

- To compile for a Cortex-A53 processor in AArch32 state, use:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a53 helloworld.c
```

- To compile for a Cortex-M4 processor, use:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m4 helloworld.c
```

- To compile for a Cortex-M33 processor, with DSP disabled, use:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m33+nodsp helloworld.c
```

- To target the AArch32 state of an Arm® Neoverse® N1 processor, use:

```
armclang --target=arm-arm-none-eabi -mcpu=neoverse-n1 helloworld.c
```

- To target the AArch64 state of an Arm Neoverse E1 processor, use:

```
armclang --target=aarch64-arm-none-eabi -mcpu=neoverse-e1 helloworld.c
```

Related information

--target (armclang)
-march (armclang)
-mcpu (armclang)
-marm (armclang)
-mthumb (armclang)
Summary of armclang command-line options

3.2 Common Arm Compiler for Embedded FuSa toolchain options

Lists the most commonly used command-line options for each of the tools in the Arm® Compiler for Embedded FuSa toolchain.

armclang common options

See the *Arm Compiler for Embedded FuSa Reference Guide* for more information about `armclang` command-line options.

Common `armclang` options include the following:

Table 3-1: armclang common options

Option	Description
-c	Performs the compilation step, but not the link step.
-x	Specifies the language of the subsequent source files, `-xc inputfile.s` or `-xc++ inputfile.s` for example.
-std	Specifies the language standard to compile for, `-std=c90` for example.
--target=arch-vendor-os-abi	Generates code for the selected Execution state (AArch32 or AArch64), for example `--target=aarch64-arm-none-eabi` or `--target=arm-arm-none-eabi`.
-march=name	Generates code for the specified architecture, for example `-march=armv8-a` or `-march=armv7-a`.
-march=list	Displays a list of all the supported architectures for the selected execution state.

Option	Description
-mcpu=name	Generates code for the specified processor, for example `-mcpu=cortex-a53`, `-mcpu=cortex-a57`, or `-mcpu=cortex-a15`.
-mcpu=list	Displays a list of all the supported processors for the selected execution state.
-marm	Requests that the compiler targets the A32 instruction set, which consists of 32-bit wide instructions only. For example, `--target=arm-arm-none-eabi -march=armv7-a -marm`. This option emphasizes performance. The `-marm` option is not valid with M-profile or AArch64 targets: • If you use the `-marm` option with an M-profile target architecture, the compiler generates an error and stops, and does not output any code. • For AArch64 targets, the compiler ignores the `-marm` option and generates a warning.
-mthumb	Requests that the compiler targets the T32 instruction set, which consists of both 16-bit wide and 32-bit wide instructions. For example, `--target=arm-arm-none-eabi -march=armv8-a -mthumb`. This option emphasizes code density. The `-mthumb` option is not valid with AArch64 targets. The compiler ignores the `-mthumb` option and generates a warning if used with AArch64 targets.
-mfloat-abi	Specifies whether to use hardware instructions or software library functions for floating-point operations.
-mfpu	Specifies the target FPU architecture.
-g (armclang)	Generates DWARF debug tables compatible with the DWARF 4 standard.
-e	Executes only the preprocessor step.
-I	Adds the specified directories to the list of places that are searched to find included files.
-o (armclang)	Specifies the name of the output file.
-Onum	Specifies the level of performance optimization to use when compiling source files.
-Os	Balances code size against code speed.
-Oz	Optimizes for code size.
-S	Outputs the disassembly of the machine code that the compiler generates.
-###	Displays diagnostic output showing the options that would be used to invoke the compiler and linker. The compilation and link steps are not performed.

armlink common options

See the *Arm Compiler for Embedded FuSa Reference Guide* for more information about `armlink` command-line options.

Common `armlink` options include the following:

Table 3-2: armlink common options

Option	Description
--scatter=filename	Creates an image memory map using the scatter-loading description that the specified file contains.
--entry	Specifies the unique initial entry point of the image.

Option	Description
--info (armlink)	Displays information about linker operation. For example, `--info=sizes,unused,unusedsymbols` displays information about all the following: • Code and data sizes for each input object and library member in the image. • Unused sections that `--remove` has removed from the code. • Symbols that were removed with the unused sections.
--list=filename	Redirects diagnostics output from options including `--info` and `--map` to the specified file.
--map	Displays a memory map containing the address and the size of each load region, execution region, and input section in the image, including linker-generated input sections.
--symbols	Lists each local and global symbol that is used in the link step, and their values.
-o filename, --output=filename	Specifies the name of the output file.
--keep=section_id	Specifies input sections that unused section elimination must not remove.
--load_addr_map_info	Includes the load addresses for execution regions and the input sections within them in the map file.

armar common options

See the *Arm Compiler for Embedded FuSa Reference Guide* for more information about `armar` command-line options.

Common `armar` options include the following:

Table 3-3: armar common options

Option	Description
--debug_symbols	Includes debug symbols in the library.
-a pos_name	Places new files in the library after the file `<pos_name>`.
-b pos_name	Places new files in the library before the file `<pos_name>`.
-a file_list	Deletes the specified files from the library.
--sizes	Lists the `Code`, `RO Data`, `RW Data`, `ZI Data`, and `Debug` sizes of each member in the library.
-t	Prints a table of contents for the library.

fromelf common options

See the *Arm Compiler for Embedded FuSa Reference Guide* for more information about `fromelf` command-line options.

Common `fromelf` options include the following:

Table 3-4: fromelf common options

Option	Description
--elf	Selects ELF output mode.
--text <options>	Displays image information in text format. The optional `<options>` specify additional information to include in the image information. Valid `<options>` include `-c` to disassemble code, and `-s` to print the symbol and versioning tables. You can also use `<options>` without specifying `--text`.

Option	Description
--info (fromelf)	Displays information about specific topics, for example `--info=totals` lists the `Code`, `RO Data`, `RW Data`, `ZI Data`, and `Debug` sizes for each input object and library member in the image.

armasm common options

See the *Arm Compiler for Embedded FuSa Reference Guide* for more information about `armasm` command-line options.

Note

Only use `armasm` to assemble legacy assembly code syntax. Use GNU syntax for new assembly files, and assemble with the `armclang` integrated assembler.

Common `armasm` options include the following:

Table 3-5: armasm common options

Option	Description
--cpu=name	Sets the target processor.
-g (armasm)	Generates DWARF debug tables compatible with the DWARF 3 standard.
--fpu=name	Selects the target floating-point unit (FPU) architecture.
-o (armasm)	Specifies the name of the output file.

3.3 Selecting source language options

`armclang` provides different levels of support for different source language standards. Arm® Compiler for Embedded FuSa infers the source language, for example C or C++, from the filename extension. You can use the `-x` and `-std` options to force Arm Compiler for Embedded FuSa to compile for a specific source language and source language standard.

Note

This topic includes descriptions of [ALPHA] and [COMMUNITY] features. See Support level definitions.

Source language

By default Arm Compiler for Embedded FuSa treats files with `.c` extension as C source files. If you want to compile a `.c` file, for example `file.c`, as a C++ source file, use the `-xc++` option:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -xc++ file.c
```

By default Arm Compiler for Embedded FuSa treats files with `.cpp` extension as C++ source files. If you want to compile a `.cpp` file, for example `file.cpp`, as a C source file, use the `-xc` option:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -xc file.cpp
```

The `-x` option only applies to input files that follow it on the command line.

Source language standard

Arm Compiler for Embedded FuSa supports Standard and GNU variants of source languages as shown in the following table:

Table 3-6: Supported C and C++ source language variants

Standard C	GNU C	Standard C++	GNU C++
c90	gnu90	c++98	gnu++98
c99	gnu99	c++03	gnu++03
c11 [COMMUNITY]	gnu11 [COMMUNITY]	c++11	gnu++11
-	-	c++14	gnu++14
-	-	c++17	gnu++17

Note

Some C and C++ language standards are supported as [COMMUNITY] features. See Support level definitions.

Note

`armclang` always applies the rules for type auto-deduction for copy-list-initialization and direct-list-initialization from C++17, regardless of which C++ source language mode a program is compiled for. For example, the compiler always deduces the type of `test` as `int` instead of `std::initializer_list<int>` in the following code:

```
auto test{ 1 };
```

The default language standard for C code is `gnu11` [COMMUNITY]. The default language standard for C++ code is `gnu++17`. To specify a different source language standard, use the `-std=<name>` option.

Compatibility of C++ compilation units

We do not guarantee the compatibility of C++ compilation units compiled with different major or minor versions of Arm Compiler for Embedded FuSa and linked into a single image. Also, the default language standards used can differ between versions of Arm Compiler for Embedded FuSa.

Note

We recommend that you always build your C++ code from source with a single version of the toolchain.

Creating and linking libraries

If you are creating libraries for third party use, we recommend that you document which version of Arm Compiler for Embedded FuSa is used to build the libraries, so that your users can ensure they are using the same version. If possible, consider providing multiple builds so that your users can select one that matches the version of the toolchain they want to use.

If you are linking your project against a pre-built library provided by a third party, ensure you use a version of the library built using the same version of the compiler toolchain you are using to build your project.

You can mix C++ with C code or C libraries.

Arm Compiler for Embedded FuSa supports various language extensions, including GCC extensions, which you can use in your source code. Some GCC extensions are only available when you specify one of the GCC C or C++ language variants. Use the `armclang` option `-Wgnu` to see if a GNU extension is used. For more information on language extensions, see the C Language Extensions in the *Arm Compiler for Embedded FuSa Reference Guide*.

Because Arm Compiler for Embedded FuSa uses the available language extensions by default, it does not adhere to the strict ISO standard. To compile to strict ISO standard for the source language, use the `-Wpedantic` option. This option generates warnings where the source code violates the ISO standard. Arm Compiler for Embedded FuSa does not support strict adherence to C++98 or C++03.

If you do not use `-Wpedantic`, Arm Compiler for Embedded FuSa uses the available language extensions without warning. However, where language variants produce different behavior, the behavior is that of the language variant that `-std` specifies.

Note

Certain compiler optimizations can violate strict adherence to the ISO standard for the language. To identify when these violations happen, use the `-Wpedantic` option.

The following example shows the use of a variable length array, which is a C99 feature. In this example, the function declares an array `i`, with variable length `<n>`.

```
#include <stdlib.h>

void function(int n) {
    int i[n];
}
```

Arm Compiler for Embedded FuSa does not warn when compiling the example for C99 with `-Wpedantic`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c -std=c99 -Wpedantic file.c
```

Arm Compiler for Embedded FuSa does warn about variable length arrays when compiling the example for C90 with `-Wpedantic`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c -std=c90 -Wpedantic file.c
```

In this case, `armclang` gives the following warning:

```
file.c:4:8: warning: variable length arrays are a C99 feature [-Wvla-extension]
int i[n];
^
1 warning generated.
```

Exceptions to language standard support

Arm Compiler for Embedded FuSa 6 with `libc++` provides varying levels of support for different source language standards. The following table lists the exceptions to the support Arm Compiler for Embedded FuSa provides for each language standard:

Table 3-7: Exceptions to the support for the language standards

Language standard	Exceptions to the support for the language standard
C90	None. C90 is fully supported.
C99	Complex numbers are not supported.
C11 [COMMUNITY]	The base Clang component provides C11 language functionality. However, Arm has performed no independent testing of these features and therefore these features are [COMMUNITY] features. Use of C11 library features is unsupported. C11 is the default language standard for C code. However, use of the new C11 language features is a community feature. Use the `-std` option to restrict the language standard if necessary. Use the `-Wc11-extensions` option to warn about any use of C11-specific features.
C++03, C++98	• The `armclang` option `-std=c++98` is an alias for `-std=c++03`. The C++03 standard is supported except: ◦ Where the C++11 standard deviates from the C++03 standard. For example, where `std::deque<T>::insert()` returns an iterator, as required by the C++11 standard, but the C++03 standard requires it to return `void`. Information about how the C++11 standard deviates from the C++03 standard is available in Annex "C Compatibility" of the C++11 standard definition. ◦ Where the `libc++` library deviates from the C++03 standard library: ▪ For `std::raw_storage_iterator`, the C++03 standard requires the `raw_storage_iterator` class template to be inherited from `std::iterator<std::output_iterator_tag,void,void,void,void>`. However, in `libc++` the `raw_storage_iterator` class template is inherited from an instantiation of `std::iterator` with a different list of template arguments. • Support for `-fno-exceptions` is limited.

Language standard	Exceptions to the support for the language standard
C++11	• Concurrency constructs or other constructs that are enabled through the following standard library headers are [ALPHA] supported: ◦ `<thread>` ◦ `<mutex>` ◦ `<shared_mutex>` ◦ `<condition_variable>` ◦ `<future>` ◦ `<chrono>` ◦ `<atomic>` For more details, contact the Arm Support team. • The C++14 sized deallocation feature is supported with C++11 if the `-fsized-deallocation` command-line option is specified.
C++14	• Concurrency constructs or other constructs that are enabled through the following standard library headers are [ALPHA] supported: ◦ `<thread>` ◦ `<mutex>` ◦ `<shared_mutex>` ◦ `<condition_variable>` ◦ `<future>` ◦ `<chrono>` ◦ `<atomic>` For more details, contact the Arm Support team. • The sized deallocation feature is supported by default for C++14. You can use the `-fno-sized-deallocation` command-line option to turn off sized deallocation.
C++17	The base Clang and `libc++` components provide C++17 language functionality. However, some features are not supported. See Standard C++ library implementation definition for more information.

Note

gnu++17 is the default language standard for C++ code.

Garbage collection support

The Arm C++ library does not support section *"Pointer safety" [util.dynamic.safety]* of the C++11, C++14, C++17, and C++20 standards. Specifically, the C++ standard library type `std::pointer_safety` and following functions and function templates are unsupported:

- `std::declare_reachable()`
- `std::undeclare_reachable()`
- `std::declare_no_pointers()`
- `std::undeclare_no_pointers()`
- `std::get_pointer_safety()`

Additional information

See the Arm Compiler for Embedded FuSa Reference Guide for information about Arm-specific language extensions.

For more information about `libc++` support, see Standard C++ library implementation definition, in the *Arm C and C++ Libraries and Floating-Point Support User Guide*.

For [COMMUNITY] supported language features, see the Clang Compiler User's Manual.

The LLVM Clang project provides the following additional information about language compatibility:

- Language compatibility:

  https://clang.llvm.org/compatibility.html
- Language extensions:

  https://releases.llvm.org/18.1.0/tools/clang/docs/LanguageExtensions.html
- C++ implementation status:

  https://clang.llvm.org/cxx_status.html

Arm Compiler for Embedded FuSa and undefined behavior

The C and C++ standards consider any code that uses non-portable, erroneous program or data constructs as undefined behavior. Arm provides no information or guarantees about the behavior of Arm Compiler for Embedded FuSa when presented with a program that exhibits undefined behavior. That includes whether the compiler attempts to diagnose the undefined behavior.

For more information about `-fsanitize=undefined` support, see -fsanitize, -fno-sanitize, in the *Arm Compiler for Embedded FuSa Reference Guide*.

Related information

Standard C++ library implementation definition
Arm Compiler for Embedded FuSa Reference Guide
-fsized-deallocation, -fno-sized-deallocation

3.4 Selecting optimization options

Arm® Compiler for Embedded FuSa performs several optimizations to reduce the code size and improve the performance of your application. There are different optimization levels that have different optimization goals. Therefore, optimizing for a certain goal has an impact on the other goals. Optimization levels are always a trade-off between these different goals.

Arm Compiler for Embedded FuSa provides various optimization levels to control the different optimization goals. The best optimization level for your application depends on your application and optimization goal.

Optimization goal	Useful optimization levels
Smaller code size	`-Oz`, `-Omin`
Faster performance	`-O2`, `-O3`, `-Ofast`, `-Omax`
Good debug experience without code bloat	`-O1`
Better correlation between source code and generated code	`-O0` (no optimization)
Faster compile and build time	`-O0` (no optimization)
Balanced code size reduction and fast performance	`-Os`

If you use a higher optimization level for performance, it has a higher impact on the other goals such as degraded debug experience, increased code size, and increased build time.

If your optimization goal is code size reduction, it has an impact on the other goals such as degraded debug experience, slower performance, and increased build time.

`armclang` provides a range of options to help you find a suitable approach for your requirements. Consider whether code size reduction or faster performance is the goal that matters most for your application, and then choose an option that matches your goal.

Optimization level -O0

`-O0` disables all optimizations. This optimization level is the default. Using `-O0` results in a faster compilation and build time, but produces slower code than the other optimization levels. Code size and stack usage are significantly higher at `-O0` than at other optimization levels. The generated code closely correlates to the source code, but significantly more code is generated, including dead code.

Optimization level -O1

`-O1` enables the core optimizations in the compiler. This optimization level provides a good debug experience with better code quality than `-O0`. Also the stack usage is improved over `-O0`. We recommend this option for a good debug experience.

The differences when using `-O1`, as compared to `-O0` are:

- Optimizations are enabled, which might reduce the accuracy, precision, or availability of debug information.
- Inlining is enabled, meaning backtraces might not give the stack of open function activations that you might expect from reading the source.
- If the result is not needed, a function with no side-effects might not be called in the expected place, or might be omitted.
- Values of variables might not be available within their scope after they are no longer used. For example, their stack location might be reused.

Optimization level -O2

`-O2` is a higher optimization for performance compared to `-O1`. It adds few new optimizations, and changes the heuristics for optimizations compared to `-O1`. This level is the first optimization level at which the compiler might automatically generate vector instructions. It also degrades the debug experience, and might result in an increased code size compared to `-O1`.

The differences when using `-O2` as compared to `-O1` are:

- The threshold at which the compiler believes that it is profitable to inline a call site might increase.
- The amount of loop unrolling that is performed might increase.
- Vector instructions might be generated for simple loops and for correlated sequences of independent scalar operations.

The creation of vector instructions can be inhibited with the `armclang` command-line option `-fno-vectorize`.

Optimization level -O3

`-O3` is a higher optimization for performance compared to `-O2`. This optimization level enables optimizations that require significant compile-time analysis and resources, and changes the heuristics for optimizations compared to `-O2`. `-O3` instructs the compiler to optimize for the performance of generated code and disregard the size of the generated code, which might result in an increased code size. It also degrades the debug experience compared to `-O2`.

The differences when using `-O3` as compared to `-O2` are:

- The threshold at which the compiler believes that it is profitable to inline a call site increases.
- The amount of loop unrolling that is performed is increased.
- More aggressive instruction optimizations are enabled late in the compiler pipeline.

Optimization level -Os

`-Os` aims to provide high performance without a significant increase in code size. Depending on your application, the performance provided by `-Os` might be similar to `-O2` or `-O3`.

`-Os` provides code size reduction compared to `-O3`. It also degrades the debug experience compared to `-O1`.

The differences when using `-Os` as compared to `-O3` are:

- The threshold at which the compiler believes it is profitable to inline a call site is lowered.
- The amount of loop unrolling that is performed is significantly lowered.

Optimization level -Oz

`-Oz` aims to provide reduced code size without using Link-Time Optimization (LTO). We recommend this option for best code size if LTO is not appropriate for your application. This optimization level degrades the debug experience compared to `-O1`.

The differences when using `-Oz` as compared to `-Os` are:

- The compiler optimizes for code size only and disregards performance optimizations, which might result in slower code.
- Function inlining is not disabled. There are instances where inlining might reduce code size overall, for example if a function is called only once. The inlining heuristics are tuned to inline only when code size is expected to decrease as a result.

- Optimizations that might increase code size, such as Loop unrolling and loop vectorization are disabled.
- Loops are generated as while loops instead of do-while loops.
- Outlining is enabled for AArch32 with M-profile and AArch64 targets only. The outliner searches for identical sequences of code and puts them in a function, then replaces each instance of the code sequence with calls to this function. Outlining reduces code size, but can increase execution time. You can override this using the -moutline, -mno-outline options.

Optimization level -Omin

`-Omin` aims to provide smaller code size than `-Oz`, by using a subset of LTO functionality. You might be able to achieve even smaller code size using `-Oz` with LTO enabled.

The differences when using `-Omin` as compared to `-Oz` are:

- `-Omin` enables a basic set of LTO aimed at removing unused code and data, while also trying to optimize global memory accesses.
- `-Omin` enables virtual function elimination, which is a particular benefit to C++ users.

If you want to compile at `-Omin` and use separate compile and link steps, then you must also include `-Omin` on your `armlink` command line.


Note

See Restrictions with Link-Time Optimization.

Optimization level -Ofast

`-Ofast` performs optimizations from level `-O3`, including those optimizations performed with the `armclang` option `-ffast-math`.

This level also performs other aggressive optimizations that might violate strict compliance with language standards.

This level degrades the debug experience, and might result in increased code size compared to `-O3`.

Optimization level -Omax

`-Omax` performs maximum optimization, and specifically targets performance optimization. It enables all the optimizations from level `-Ofast`, together with LTO.

At this optimization level, Arm Compiler for Embedded FuSa might violate strict compliance with language standards. Use this optimization level for the fastest performance.

This level degrades the debug experience, and might result in increased code size compared to `-Ofast`.

If you want to compile at `-Omax` and have separate compile and link steps, then you must also include `-Omax` on your `armlink` command line.

Note

See Restrictions with Link-Time Optimization.

Example: C source code

Create the file `file.c` containing the following C code:

```
int test()
{
    int x=10, y=20;
    int z;
    z=x+y;
    return 0;
}
```

The source file contains mostly dead code, such as `int x=10` and `z=x+y`. In the following examples:

- At optimization level `-O0`, the compiler performs no optimization, and therefore generates code for the dead code in the source file.
- At optimization level `-O1`, the compiler does not generate code for the dead code in the source file.

Example: Code generation with -O0

Compile the C source file with the `-O0` optimization option:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O0 -S file.c
```

The unoptimized output from `armclang` is:

```
test:
    .fnstart
    .pad #12
    sub     sp, sp, #12
    mov     r0, #10
    str     r0, [sp, #8]
    mov     r0, #20
    str     r0, [sp, #4]
    ldr     r0, [sp, #8]
    add     r0, r0, #20
    str     r0, [sp]
    mov     r0, #0
    add     sp, sp, #12
    bx      lr
```

Example: Code generation with -O1

Compile the C source file with the `-O1` optimization option:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O1 -S file.c
```

The optimized output from `armclang` is:

```
test:
    .fnstart
    movs r0, #0
    bx lr
```

Related information

Optimizing for code size or performance on page 95
Optimizing loops on page 77
Optimizing across modules with Link-Time Optimization on page 98
-O

3.5 Building to aid debugging

During application development, you must debug the image that you build. The Arm® Compiler for Embedded FuSa tools have various features that provide good debug view and enable source-level debugging, such as setting breakpoints in C and C++ code. There are also some features you must avoid when building an image for debugging.

Available command-line options

To build an image for debugging, you must compile with the `-g` option. This option allows you to specify the DWARF format to use. The `-g` option is a synonym for `-gdwarf-4`. You can specify DWARF 2, DWARF 3, or DWARF 5 if necessary, for example:

```
armclang -gdwarf-3
```

When linking, there are several `armlink` options available to help improve the debug view:

- `--debug`. This option is the default.
- `--no_remove` to retain all input sections in the final image even if they are unused.
- `--bestdebug`. When different input objects are compiled with different optimization levels, this option enables linking for the best debug illusion.

Effect of optimizations on the debug view

To build an application that gives the best debug view, it is better to use options that give the fewest optimizations. We recommend using optimization level `-O1` for debugging. This option gives good code density with a satisfactory debug view.

Higher optimization levels perform progressively more optimizations with correspondingly poorer debug views.

The compiler attempts to automatically inline functions at all optimization levels except at `-O0`. However, the threshold at which the compiler decides to inline depends on the level. If you must use optimization levels higher than `-O0`, disable the automatic inlining with the `armclang` option `-fno-inline-functions`. The linker inlining is disabled by default.

Support for debugging overlaid programs

The linker provides various options to support overlay-aware debuggers:

- `--emit_debug_overlay_section`
- `--emit_debug_overlay_relocs`

These options permit an overlay-aware debugger to track which overlay is active.

Features to avoid when building an image for debugging

Avoid using the following in your source code:

- The `__attribute__((always_inline))` function attribute. Qualifying a function with this attribute forces the compiler to inline the function. If you also use the `-fno-inline-functions` option, the function is inlined.
- The `__declspec(noreturn)` attribute and the `__attribute__((noreturn))` function attribute. These attributes limit the ability of a debugger to display the call stack.

Avoid using the following features when building an image for debugging:

- Link-Time Optimization. This feature performs aggressive optimizations and can remove large chunks of code.
- The `armlink` option `--no_debug`.
- The `armlink` option `--inline`. This option changes the image in such a way that the debug information might not correspond to the source code.

3.6 Linking object files to produce an executable

The linker combines the contents of one or more object files with selected parts of any required object libraries to produce executable images, partially linked object files, or shared object files.

The command for invoking the linker is:

```
armlink <options> <input-file-list>
```

where:

<options>
are linker command-line options.

<input-file-list>
is a space-separated list of objects, libraries, or symbol definitions (symdefs) files.

For example, to link the object file `hello_world.o` into an executable image `hello_world.axf`:

```
armlink -o hello_world.axf hello_world.o
```

Compatibility of object files

We do not guarantee the compatibility of C++ compilation units compiled with different major or minor versions of Arm® Compiler for Embedded FuSa and linked into a single image. Therefore, we recommend that you always build your C++ code from source with a single version of the toolchain.

3.7 Linker options for mapping code and data to target memory

For an image to run correctly on a target, you must place the various parts of the image at the correct locations in memory. Linker command-line options are available to map the various parts of an image to target memory.

The options implement the scatter-loading mechanism that describes the memory layout for the image. The options that you use depend on the complexity of your image:

- For simple images, use the following memory map related options:
  - `--ro_base` to specify the address of both the load and execution region containing the RO output section.
  - `--rw_base` to specify the address of the execution region containing the RW output section.
  - `--zi_base` to specify the address of the execution region containing the ZI output section.


Note

For objects that include eXecute-Only (XO) sections, the linker provides the `--xo_base` option to locate the XO sections. These sections are objects that are targeted at Arm®v6-M, Armv7-M, or Armv8-M architectures, or objects that are built with the `armclang` option `-mthumb`. However, XO is not supported on Armv6-M for any form of position independent code.

- For complex images, use a text format scatter-loading description file. This file is known as a scatter file, and you specify it with the `--scatter` option.


Note

You cannot use the memory map related options with the `--scatter` option.

Examples

The following example shows how to place code and data using the memory map related options:

```
armlink --ro_base=0x0 --rw_base=0x400000 --zi_base=0x405000 --first="init.o(init)"
 init.o main.o
```

Note

In this example, `--first` is also included to make sure that the initialization routine is executed first.

The following example shows a scatter file, `scatter.scat`, that defines an equivalent memory map:

```
LR1 0x0000 0x20000
{
    ER_RO 0x0
    {
        init.o (INIT, +FIRST)
        *(+RO)
    }

    ER_RW 0x400000
    {
        *(+RW)
    }

    ER_ZI 0x405000
    {
        *(+ZI)
    }
}
```

To link with this scatter file, use the following command:

```
armlink --scatter=scatter.scat init.o main.o
```

3.8 Passing options from the compiler to the linker

By default, when you run `armclang` the compiler automatically invokes the linker, `armlink`.

A number of `armclang` options control the behavior of the linker. These options are translated to equivalent `armlink` options.

Table 3-9: armclang linker control options

armclang Option	armlink Option	Description
`-e`	`--entry`	Specifies the unique initial entry point of the image.
`-L`	`--userlibpath`	Specifies a list of paths that the linker searches for user libraries.
`-l`	`--library`	Add the specified library to the list of searched libraries.
`-u`	`--undefined`	Prevents the removal of a specified symbol if it is undefined.

In addition, the `-Xlinker` and `-Wl` options let you pass options directly to the linker from the compiler command line. These options perform the same function, but use different syntaxes:

- The `-Xlinker` option specifies a single option, a single argument, or a single `option=argument` pair. If you want to pass multiple options, use multiple `-Xlinker` options.
- The `-Wl,` option specifies a comma-separated list of options and arguments or `option=argument` pairs.

For example, the following are all equivalent because `armlink` treats the single option `--list=diag.txt` and the two options `--list diag.txt` equivalently:

```
-Xlinker --list -Xlinker diag.txt -Xlinker --split

-Xlinker --list=diag.txt -Xlinker --split

-Wl,--list,diag.txt,--split

-Wl,--list=diag.txt,--split
```


Note

The `-###` compiler option produces diagnostic output showing exactly how the compiler and linker are invoked, displaying the options for each tool. With the `-###` option, `armclang` only displays this diagnostic output. It does not compile source files or invoke `armlink`.

The following example shows how to use the `-Xlinker` option to pass the `--split` option to the linker, splitting the default load region containing the RO and RW output sections into separate regions:

```
armclang hello.c --target=aarch64-arm-none-eabi -Xlinker --split
```

You can use `fromelf --text` to compare the differences in image content:

```
armclang hello.c --target=aarch64-arm-none-eabi -o hello_DEFAULT.axf
armclang hello.c --target=aarch64-arm-none-eabi -o hello_SPLIT.axf -Xlinker --split

fromelf --text hello_DEFAULT.axf > hello_DEFAULT.txt
fromelf --text hello_SPLIT.axf > hello_SPLIT.txt
```

3.9 Controlling diagnostic messages

Arm® Compiler for Embedded FuSa provides diagnostic messages in the form of warnings and errors. You can use options to suppress these messages or enable them as either warnings or errors.

Arm Compiler for Embedded FuSa lists all the warnings and errors it encounters during the compiling and linking process.

Message format for armclang

`armclang` produces messages in the following format:

`:<file>:<line>:<col>: <type>: <message>`

`<file>`
The filename that contains the error or warning.

`<line>`
The line number that contains the error or warning.

`<col>`
The column number that generated the message.

`<type>`
The type of the message, for example error or warning.

`<message>`
The message text. This text might end with a diagnostic flag of the form `-W<flag>`, for example `-Wvla-extension`, to identify the error or warning. Only the messages that you can suppress have an associated flag. Errors that you cannot suppress do not have an associated flag.

An example warning diagnostic message is:

```
file.c:8:7: warning: variable length arrays are a C99 feature [-Wvla-extension]
 int i[n];
      ^
```

This warning message tells you:

- The file that contains the problem is called `file.c`.
- The problem is on line 8 of `file.c`, and starts at character 7.
- The warning is about the use of a variable length array `i[n]`.
- The flag to identify, enable, or disable this diagnostic message is `vla-extension`.

The following are common options that control diagnostic output from `armclang`.

Table 3-10: Common diagnostic options

Option	Description
`-Werror`	Turn all warnings into errors.
`-Werror=<flag>`	Turn warning flag `<flag>` into an error.
`-Wno-error=<flag>`	Leave warning flag `<flag>` as a warning even if `-Werror` is specified.
`-W<flag>`	Enable warning flag `<flag>`.
`-Wno-<flag>`	Suppress warning flag `<flag>`.
`-w`	Suppress all warnings. Note that this option is a lowercase `w`.
`-Weverything`	Enable all warnings.
`-Wpedantic`	Generate warnings if code violates strict ISO C and ISO C++.
`-pedantic`	Generate warnings if code violates strict ISO C and ISO C++.
`-pedantic-errors`	Generate errors if code violates strict ISO C and ISO C++.

See *Options to Control Error and Warning Messages* in the Clang Compiler User's Manual for full details about controlling diagnostics with `armclang` and for possible values for `<flag>`.


Note

The open-source documentation at https://releases.llvm.org/18.1.0/docs/ and https://releases.llvm.org/18.1.0/tools/clang/docs is beyond the control of Arm, but it should be generally well aligned with Arm Compiler for Embedded FuSa 6.22.2.

Examples: Controlling diagnostic messages with armclang

Copy the following code example to `file.c` and compile it with Arm Compiler for Embedded FuSa to see example diagnostic messages.

```
#include <stdlib.h>
#include <stdio.h>

void function (int x) {
  int i;
  int y=i+x;

  printf("Result of %d plus %d is %d\n", i, x); /* Missing an input argument for the
 third %d */
  call(); /* This function has not been declared and is therefore an implicit
 declaration */

  return;
}
```

Compile `file.c` using:

```
armclang --target=aarch64-arm-none-eabi -march=armv8 -c file.c
```

By default, `armclang` checks the format of `printf()` statements to ensure that the number of `%` format specifiers matches the number of data arguments. By default, `armclang` also compiles for the `gnu11` standard for `.c` files. This language standard does not allow implicit function declarations. Therefore, `armclang` generates the following diagnostic messages:

```
file.c:8:38: warning: more '%' conversions than data arguments [-Wformat-
insufficient-args]
    8 |     printf("Result of %d plus %d is %d\n", i, x); /* Missing an input
 argument for the third %d */
      |                                          ~^
file.c:9:5: error: call to undeclared function 'call'; ISO C99 and later do not
 support implicit function declarations
      [-Wimplicit-function-declaration]
    9 |     call(); /* This function has not been declared and is therefore an
 implicit declaration */
      |     ^
1 warning and 1 error generated.
```

To suppress all warnings, use `-w`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c file.c -w
```

To suppress only the `-Wformat` warning, use `-Wno-format`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c file.c -Wno-format
```

To enable the `-Wformat` message as an error, use `-Werror=format`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c file.c -Werror=format
```

Some diagnostic messages are suppressed by default. To see all diagnostic messages, use `-Weverything`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -c file.c -Weverything
```

Pragmas for controlling diagnostics with armclang

Pragmas within your source code can control the output of diagnostics from the `armclang` compiler.

See *Controlling Diagnostics via Pragmas* in the Clang Compiler User's Manual for full details about controlling diagnostics with `armclang`.

The following are some of the common options that control diagnostics:

`#pragma clang diagnostic ignored "-W<name>"`
Ignores the diagnostic message specified by `<name>`.

`#pragma clang diagnostic warning "-W<name>"`
Sets the diagnostic message specified by `<name>` to warning severity.

`#pragma clang diagnostic error "-W<name>"`
Sets the diagnostic message specified by `<name>` to error severity.

`#pragma clang diagnostic fatal "-W<name>"`
Sets the diagnostic message specified by `<name>` to fatal error severity.

`#pragma clang diagnostic push`
Saves the diagnostic state so that it can be restored.

`#pragma clang diagnostic pop`
Restores the last saved diagnostic state.

The compiler provides appropriate diagnostic names in the diagnostic output.


Note

Alternatively, you can use the command-line option, `-W<name>`, to suppress or change the severity of messages, but the change applies for the entire compilation.

Example: Use of pragmas to selectively override a command-line option

`file1.c`:

```
#if file1
#endif file1 /* no warning when compiling with -Wextra-tokens */

#pragma clang diagnostic push
#pragma clang diagnostic warning "-Wextra-tokens"

#if file1
#endif file1 /* warning: extra tokens at end of #endif directive */

#pragma clang diagnostic pop
```

Compile this example with:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -c file1.c -o file1.o -Wno-extra-
tokens
```

The compiler only generates a warning for the second instance of `#endif file1`:

```
file1.c:8:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]

#endif file1 /* warning: extra tokens at end of #endif directive */
       ^
       //
1 warning generated.
```

Message format for other tools

The other tools in the toolchain, such as `armasm` and `armlink`, produce messages in the following format:

```
type: prefix id suffix: message_text
```

`<type>`

One of the following types:

Internal fault

Internal faults indicate an internal problem with the tool. Contact your supplier with feedback.

Error

Errors indicate problems that cause the tool to stop.

Warning

Warnings indicate unusual conditions that might indicate a problem, but the tool continues.

Remark

Remarks indicate common, but sometimes unconventional, tool usage. These diagnostics are not displayed by default. The tool continues.

<prefix>
The tool that generated the message, one of:

- `A` - `armasm`
- `L` - `armlink` or `armar`
- `Q` - `fromelf`

<id>
A unique numeric message identifier.

<suffix>
The type of message, one of:

- `E` - Error
- `W` - Warning
- `R` - Remark

<message_text>
The text of the message.

For example, the following `armlink` error message:

```
Error: L6449E: While processing /home/scratch/a.out: I/O error writing file '/home/
scratch/a.out': Permission denied
```

All the diagnostic messages that are in this format, and any additional information, are in the Arm Compiler for Embedded FuSa Errors and Warnings Reference Guide.

Options for controlling diagnostics with the other tools

Several different options control diagnostics with the `armasm`, `armlink`, `armar`, and `fromelf` tools:

`--brief_diagnostics`
`armasm` only. Uses a shorter form of the diagnostic output. The original source line is not displayed and the error message text is not wrapped when it is too long to fit on a single line.

`--diag_error=<tag>[,<tag>]...`
Sets the specified diagnostic messages to Error severity. Use `--diag_error=warning` to treat all warnings as errors.

`--diag_remark=<tag>[,<tag>]...`
Sets the specified diagnostic messages to Remark severity.

`--diag_style=arm|ide|gnu`
Specifies the display style for diagnostic messages.

`--diag_suppress=<tag>[,<tag>]...`
Suppresses the specified diagnostic messages. Use `--diag_suppress=error` to suppress all errors that can be downgraded, or `--diag_suppress=warning` to suppress all warnings.

Reducing the severity of diagnostic messages might prevent the tool from reporting important faults. We recommend that you do not reduce the severity of diagnostics unless you understand the impact on your software.

`--diag_warning=<tag>[,<tag>]...`

Sets the specified diagnostic messages to Warning severity. Use `--diag_warning=error` to set all errors that can be downgraded to warnings.

`--errors=<filename>`

Redirects the output of diagnostic messages to the specified file.

`--remarks`

`armlink` only. Enables the display of remark messages (including any messages redesignated to remark severity using `--diag_remark`).

`<tag>` is the four-digit diagnostic number, `<nnnn>`, with the tool letter prefix, but without the letter suffix indicating the severity. A full list of tags with the associated suffixes is in the Arm Compiler for Embedded FuSa Errors and Warnings Reference Guide.

For example, to downgrade a warning message to Remark severity:

Create the file `noend.s` containing:

```
    AREA ||.text||,CODE

x   EQU 42
    IF :LNOT: :DEF: sym
        ASSERT x == 42
    ENDIF
sym EQU 1
    ;END    ; Commented out
```

Assemble the file with the following commands:

```
$  armasm noend.s --cpu=8-A.32
Warning: A1950W: The legacy armasm assembler is deprecated. Consider using the
 armclang integrated assembler instead.
"noend.s", line 9: Warning: A1313W: Missing END directive at end of file
    9 00000000
0 Errors, 2 Warnings

$ armasm noend.s --cpu=8-A.32 --diag_remark=A1313,A1950
The legacy armasm assembler is deprecated. Consider using the armclang integrated
 assembler instead.
"noend.s", line 9: Missing END directive at end of file
```

Related information

-W (armclang)
The LLVM Compiler Infrastructure Project
Clang Compiler User's Manual

3.10 Selecting floating-point options

Arm® Compiler for Embedded FuSa supports floating-point arithmetic and floating-point data types in your source code or application.

Arm Compiler for Embedded FuSa supports floating-point arithmetic by using one of the following:

- Libraries that implement floating-point arithmetic in software.
- Hardware floating-point registers and instructions that are available on most Arm-based processors.

You can use various options that determine how Arm Compiler for Embedded FuSa generates code for floating-point arithmetic. Depending on your target, you might need to specify one or more of these options to generate floating-point code that correctly uses floating-point hardware or software libraries.

Table 3-11: Options for floating-point selection

Option	Description
`armclang -mfpu`	Specify the floating-point architecture to the compiler. This option is ignored with AArch64 targets.
`armclang -mfloat-abi`	Specify the floating-point linkage to the compiler.
`armclang -march`	Specify the target architecture to the compiler. This option automatically selects the default floating-point architecture.
`armclang -mcpu`	Specify the target processor to the compiler. This option automatically selects the default floating-point architecture.
`armlink --fpu`	Specify the floating-point architecture to the linker.

To improve performance, the compiler can use floating-point registers instead of the stack. You can disable this feature with the [COMMUNITY] option `-mno-implicit-float`.


Note

Avoid specifying both the architecture (`-march`) and the processor (`-mcpu`) because specifying both has the potential to cause a conflict. The compiler infers the correct architecture from the processor.

- If you want to run code on one particular processor, specify the processor using `-mcpu`. Performance is optimized, but code is only guaranteed to run on that processor. If you specify a value for `-mcpu`, do not also specify a value for `-march`.
- If you want your code to run on a range of processors from a particular architecture, specify the architecture using `-march`. The code runs on any processor implementation of the target architecture, but performance might be impacted. If you specify a value for `-march`, do not also specify a value for `-mcpu`.


Note

The `-mfpu` option is ignored with AArch64 targets, for example `aarch64-arm-none-eabi`. Use the `-mcpu` option to override the default FPU for `aarch64-arm-none-eabi` targets. For example, to prevent the use of floating-point instructions or floating-point registers for the `aarch64-arm-none-eabi` target use the `-mcpu=name+nofp+nosimd` option. Subsequent use of floating-point data types in this mode is unsupported.

Benefits of using floating-point hardware versus software floating-point libraries

Code that uses floating-point hardware is more compact and faster than code that uses software libraries for floating-point arithmetic. But code that uses the floating-point hardware can only be run on processors that have the floating-point hardware. Code that uses software floating-point libraries can run on Arm-based processors that do not have floating-point hardware, for example the Cortex®-M0 processor. Therefore, using software floating-point libraries makes the code more portable. You might also disable floating-point hardware to reduce power consumption.

Enabling and disabling the use of floating-point hardware

By default, Arm Compiler for Embedded FuSa uses the available floating-point hardware that is based on the target you specify for `-mcpu` or `-march`. However, you can force Arm Compiler for Embedded FuSa to disable the floating-point hardware. Disabling floating-point hardware forces Arm Compiler for Embedded FuSa to use software floating-point libraries, if available, to perform the floating-point arithmetic in your source code.

When compiling for AArch64:

- By default, Arm Compiler for Embedded FuSa uses floating-point hardware that is available on the target.
- To disable the use of floating-point arithmetic, use the `+nofp` extension on the `-mcpu` or `-march` options.

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a+nofp
```

- Disabling floating-point arithmetic does not disable all the floating-point hardware because the floating-point hardware is also used for Advanced Single Instruction Multiple Data (SIMD) arithmetic. To disable all Advanced SIMD and floating-point hardware, use the `+nofp+nosimd` extension on the `-mcpu` or `-march` options:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a+nofp+nosimd
```

See -march (armclang) and -mcpu (armclang) in the *Arm Compiler For Embedded FuSa Reference Guide* for more information.

When compiling for AArch32:

- By default, Arm Compiler for Embedded FuSa uses floating-point hardware that is available on the target, except for Armv6-M, which does not have any floating-point hardware.

- To disable the use of floating-point hardware instructions, use the `-mfpu=none` option.

```
armclang --target=arm-arm-none-eabi -march=armv8-a -mfpu=none
```

- On AArch32 targets, using `-mfpu=none` disables the hardware for both Advanced SIMD and floating-point arithmetic. You can use `-mfpu` to selectively enable certain hardware features. For example, if you want to use the hardware for Advanced SIMD operations on an Armv7 architecture-based processor, but not for floating-point arithmetic, then use `-mfpu=neon`.

```
armclang --target=arm-arm-none-eabi -march=armv7-a -mfpu=neon
```

- The Armv8.1-M architecture profile has optional support for the M-profile Vector Extension (MVE). `-march` and `-mcpu` support certain MVE floating-point combinations.

```
armclang --target=arm-arm-none-eabi -march=armv8.1-m.main+mve.fp
```

See -march (armclang), -mcpu (armclang), and -mfpu (armclang) in the *Arm Compiler For Embedded FuSa Reference Guide* for more information.

Floating-point linkage

Floating-point linkage refers to how the floating-point arguments are passed to and returned from function calls.

For AArch64, you can use the `-mabi=<name>` option to specify the calling convention.

For AArch32, Arm Compiler for Embedded FuSa can use hardware linkage or software linkage. When using software linkage, Arm Compiler for Embedded FuSa passes and returns floating-point values in general-purpose registers. By default, Arm Compiler for Embedded FuSa uses software linkage. You can use the `-mfloat-abi` option to force hardware linkage or software linkage.

Table 3-12: Floating-point linkage for AArch32

-mfloat-abi value	Linkage	Floating-point operations
`hard`	Hardware linkage. Use floating-point registers. But if `-mfpu=none` is specified for AArch32, then use general-purpose registers.	Use hardware floating-point instructions. But if `-mfpu=none` is specified for AArch32, then use software libraries.
`soft`	Software linkage. Use general-purpose registers.	Use software libraries without floating-point hardware.
`softfp` (This value is the default)	Software linkage. Use general-purpose registers.	Use hardware floating-point instructions. But if `-mfpu=none` is specified for AArch32, then use software libraries.

Code with hardware linkage can be faster than the same code with software linkage. However, code with software linkage can be more portable because it does not require the hardware floating-point registers. Hardware floating-point is not available on some architectures such as Armv6-M, or on processors where the floating-point hardware might be powered down for energy efficiency reasons.

Note

In AArch32 state, if you specify `-mfloat-abi=soft`, then specifying the `-mfpu` option does not have an effect.

See the *Arm Compiler For Embedded FuSa Reference Guide* for more information on the -mfloat-abi option.

Note

All objects to be linked together must have the same type of linkage. If you link object files that have hardware linkage with object files that have software linkage, then the image might have unpredictable behavior. When linking objects, specify the `armlink` option `--fpu=<name>` where `<name>` specifies the correct linkage type and floating-point hardware. This option enables the linker to provide diagnostic information if it detects different linkage types.

See the *Arm Compiler For Embedded FuSa Reference Guide* for more information on how the --fpu=name (armlink) option specifies the linkage type and floating-point hardware.

Related information

-mabi=<name> (armclang)
-march (armclang)
-mcpu (armclang)
-mfloat-abi
-mfpu (armclang)
Floating-point support

3.11 Compilation tools command-line option rules

You can use command-line options to control many aspects of the compilation tools' operation. There are rules that apply to each tool.

armclang option rules

`armclang` follows the same syntax rules as GCC. Some options are preceded by a single dash `-`, others by a double dash `--`. Some options require an `=` character between the option and the argument, others require a space character.

armasm, armar, armlink, and fromelf command-line syntax rules

The following rules apply, depending on the type of option:

Single-letter options

All single-letter options, including single-letter options with arguments, are preceded by a single dash `-`. You can use a space between the option and the argument, or the argument can immediately follow the option. For example:

```
armar -r -a obj1.o mylib.a obj2.o

armar -r -aobj1.o mylib.a obj2.o
```

Keyword options

All keyword options, including keyword options with arguments, are preceded by a double dash `--`. An `=` or space character is required between the option and the argument. For example:

```
armlink myfile.o --cpu=list

armlink myfile.o --cpu list
```

Command-line syntax rules common to all tools

To compile files with names starting with a dash, use the POSIX option `--` to specify that all subsequent arguments are treated as filenames, not as command switches. For example, to link a file named `-ifile_1`, use:

```
armlink -- -ifile_1
```

In some Unix shells, you might have to include quotes when using arguments to some command-line options, for example:

```
armlink obj1.o --keep="s.o(vect)"
```

4. Writing Optimized Code

To make the best use of the optimization capabilities of Arm® Compiler for Embedded FuSa, there are various options, pragmas, attributes, and coding techniques that you can use.

4.1 Effect of the volatile keyword on compiler optimization

Use the `volatile` keyword when declaring variables that the compiler must not optimize. If you do not use the `volatile` keyword where it is needed, then the compiler might optimize accesses to the variable and generate unintended code or remove intended functionality.

What volatile means

The declaration of a variable as `volatile` tells the compiler that the variable can be modified at any time by another entity that is external to the implementation, for example:

- By the operating system.
- By hardware.

This declaration ensures that the compiler does not optimize any use of the variable on the assumption that this variable is unused or unmodified.

You can also use `volatile` to tell the compiler that a block containing inline assembly code has side-effects that the output, input, and clobber lists do not represent.


Note

Arm® Compiler for Embedded FuSa does not guarantee that a single-copy atomic instruction is used to access a `volatile` variable that is larger than the natural architecture data size, even when one is available for the target processor. For more information, see Volatile variables and *Atomicity in the Arm architecture* in the following documents:

- Arm Architecture Reference Manual for A-profile architecture.
- ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition.

When to use volatile

Use the `volatile` keyword for variables that might be modified from outside the scope where they are defined. Some examples are:

- If the program uses a global variable in some computation, the compiler generates code to load the value of the variable into a register to perform that computation. If the same global variable is subsequently used in another computation, the compiler might reuse the existing value in the register instead of generating another load. This reuse is because the optimizer assumes that non-volatile variables cannot be modified externally, and this assumption is not correct for memory-mapped peripherals. See Example: Infinite loop when not using the volatile keyword and Example: Infinite loop when using the volatile keyword.

- A variable might be used to implement a sleep or timer delay. If the variable appears unused, the compiler might remove the timer delay code, unless the variable is declared as `volatile`.
- In C++, an interrupt function might be defined in a `class` scope but is called by hardware asynchronously. A buffer, `buffer_full`, is modified in an interrupt and is in a scope but must still be declared as `volatile`, for example:

```
class myclass
{
    public:
    int check_stream();
    void async_interrupt();
    private:
    bool buffer_full;  // must be declared as volatile
};
int myclass::check_stream()
{
    int count = 0;
    while (!buffer_full)
    {
        count++;
    }
    return count;
}
void myclass::async_interrupt()
{
    buffer_full = !buffer_full;
}
```

In practice:

- We recommend that you declare the variables that you use to access memory-mapped peripherals as `volatile`. Even with the minimum optimization level `-O0`, there is no guarantee that a non-volatile variable is not going to be optimized.
- `volatile` is not a means of inter-thread communication or synchronization, and atomics must be used for this purpose instead. That is:
  - The `_Atomic` qualifier and `<stdatomic.h>` functions in C.
  - The `<atomic>` library functions and templates in C++.
- Interrupt and signal handlers must use either atomics or variables of the type `volatile sig_atomic_t`, but not arbitrary `volatile`-qualified types, to synchronize with other threads of execution.

Also consider using `volatile` before any inline assembly code.

Potential problems when not using volatile

When a volatile variable is not declared as `volatile`, the compiler assumes that its value cannot be modified from outside the scope that it is defined in. Therefore, the compiler might perform unwanted optimizations. This problem can manifest itself in various ways:

- Code might become stuck in a loop while polling hardware.
- Optimization might result in the removal of code that implements deliberate timing delays.
- If your code contains the infinite loop `for (;;);`, the compiler optimization might remove the loop. For more information about optimizing infinite loops, see *Infinite loops* in Optimizing loops.

Forcing the use of a specific instruction to access memory

Specifying a variable as `volatile` does not guarantee that any particular machine instruction is used to access it. For example, the AXI peripheral port on Cortex®-R7 and Cortex-R8 is a 64-bit peripheral register. This register must be written to using a two-register `STM` instruction, and not by either an `STRD` instruction or a pair of `STR` instructions. There is no guarantee that the compiler selects the access method required by that register in response to a `volatile` modifier on the associated variable or pointer type.

If you are writing code that must access the AXI port, or any other memory-mapped location that requires a particular access strategy, then declaring the location as a `volatile` variable is not enough. You must also perform your accesses to the register using an `__asm__` statement containing the load or store instructions you need. For example:

```
__asm__ volatile("stm %1,{%Q0,%R0}" : : "r"(val), "r"(ptr));
__asm__ volatile("ldm %1,{%Q0,%R0}" : "=r"(val) : "r"(ptr));
```

Example: Infinite loop when not using the volatile keyword

Create the file `read_stream.c` for a nonvolatile version of a buffer loop:

```
int buffer_full;
int read_stream(void)
{
    int count = 0;
    while (!buffer_full)
    {
        count++;
    }
    return count;
}
```

The routine increments a counter in a loop until a status flag `buffer_full` is set to true. The state of `buffer_full` can change asynchronously with program flow.

This example does not declare the variable `buffer_full` as `volatile` and is therefore wrong.

Compile the `read_stream.c` file with:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -Os -S read_stream.c
```

The disassembly in `read_stream.s` for the nonvolatile version of buffer loop contains:

```
read_stream:
        movw    r0, :lower16:buffer_full
        movt    r0, :upper16:buffer_full
        ldr     r1, [r0]
        mvn     r0, #0
.LBB0_1:
        add     r0, r0, #1
        cmp     r1, #0
        beq     .LBB0_1     ; infinite loop
        bx      lr
```

In the disassembly of the nonvolatile example, the statement `LDR r1, [r0]` loads the value of `buffer_full` into register `r1` outside the loop labeled `.LBB0_1`. Because `buffer_full` is not declared as `volatile`, the compiler assumes that its value cannot be modified outside the program. Having already read the value of `buffer_full` into `r0`, the compiler omits reloading the variable when optimizations are enabled, because its value cannot change. The result is the infinite loop labeled `.LBB0_1`.

Example: Infinite loop when using the volatile keyword

Create the file `read_stream.c` for a volatile version of a buffer loop:

```
volatile int buffer_full;
int read_stream(void)
{
    int count = 0;
    while (!buffer_full)
    {
        count++;
    }
    return count;
}
```

The routine increments a counter in a loop until a status flag `buffer_full` is set to true. The state of `buffer_full` can change asynchronously with program flow.

This example declares the variable `buffer_full` as `volatile`.

Compile the `read_stream.c` file with:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -Os -S read_stream.c
```

The disassembly in `read_stream.s` for the volatile version of buffer loop contains:

```
read_stream:
        movw    r1, :lower16:buffer_full
        mvn     r0, #0
        movt    r1, :upper16:buffer_full
.LBB1_1:
        ldr     r2, [r1]      ; buffer_full
        add     r0, r0, #1
        cmp     r2, #0
        beq     .LBB1_1
        bx      lr
```

In the disassembly of the volatile example, the compiler assumes that the value of `buffer_full` can change outside the program and performs no optimization. Therefore, the value of `buffer_full` is loaded into register `r2` inside the loop labeled `.LBB1_1`. As a result, the assembly code that is generated for loop `.LBB1_1` is correct.

Related information

Floating-point division by zero errors in C and C++ code on page 280
Volatile variables
armclang Inline Assembler

Arm Cortex-R7 MPCore Technical Reference Manual
Arm Cortex-R8 MPCore Processor Technical Reference Manual

4.2 Optimizing loops

Loops can take a significant amount of time to complete depending on the number of iterations in the loop. The overhead of checking a condition for each iteration of the loop can degrade the performance of the loop.

Loop unrolling

You can reduce the impact of this overhead by unrolling some of the iterations, which in turn reduces the number of iterations for checking the condition. Use `#pragma unroll (<n>)` to unroll time-critical loops in your source code. However, unrolling loops has the disadvantage of increasing the code size. These pragmas are only effective at optimization `-O2`, `-O3`, `-Ofast`, and `-Omax`.

Table 4-1: Loop unrolling pragmas

Pragma	Description
`#pragma unroll (<n>)`	Unroll <n> iterations of the loop.
`#pragma unroll_completely`	Unroll all the iterations of the loop.

Note

Manually unrolling loops in source code might hinder the automatic rerolling of loops and other loop optimizations by the compiler. We recommend that you use `#pragma unroll` instead of manually unrolling loops. See #pragma unroll[(n)], #pragma unroll_completely in the *Arm Compiler for Embedded FuSa Reference Guide* for more information.

The following examples show code with loop unrolling and code without loop unrolling:

Bit counting loop without unrolling

Create the file `file.c` containing:

```
int countSetBits1(unsigned int n)
{
    int bits = 0;

    while (n != 0)
    {
        if (n & 1) bits++;
        n >>= 1;
    }
    return bits;
}
```

Compile with:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O2 -S file.c -o file.s
```

The disassembly in `file.s` contains:

```
countSetBits1:
   ...
    cmp r0, #0
    moveq   r0, #0
    bxeq    lr
.LBB0_1:
    mov r1, r0
    mov r0, #0
.LBB0_2:                        @ =>This Inner Loop Header: Depth=1
    and r2, r1, #1
    lsrs    r1, r1, #1
    add r0, r0, r2
    bne .LBB0_2
@ %bb.3:
    bx  lr
```

Bit counting loop with unrolling

Copy the following into the file `unroll.c`:

```
int countSetBits2(unsigned int n)
{
    int bits = 0;
    #pragma unroll (4)
    while (n != 0)
    {
        if (n & 1) bits++;
        n >>= 1;
    }
    return bits;
}
```

Compile with:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O2 -S unroll.c -o unroll.s
```

The disassembly in `unroll.s` contains:

```
countSetBits1:
   ...
    cmp r0, #0
    moveq   r0, #0
    bxeq    lr
.LBB0_1:
    mov r1, r0
    mov r0, #0
    b   .LBB0_3
.LBB0_2:                        @   in Loop: Header=BB0_3 Depth=1
    and r2, r2, #1
    lsrs    r1, r1, #4
    add r0, r0, r2
    bxeq    lr
.LBB0_3:                        @ =>This Inner Loop Header: Depth=1
    and r2, r1, #1
    add r0, r0, r2
    lsrs    r2, r1, #1
    beq .LBB0_5
@ %bb.4:                        @   in Loop: Header=BB0_3 Depth=1
    and r2, r2, #1
    add r0, r0, r2
    lsrs    r2, r1, #2
```

```
    andne   r2, r2, #1
    addne   r0, r0, r2
    lsrsne  r2, r1, #3
    bne .LBB0_2
.LBB0_5:
    bx  lr
```

In this example, the generated code is faster but larger in size.

Arm® Compiler for Embedded FuSa can unroll loops completely only if the number of iterations is known at compile time.

Loop vectorization

If your target has the Advanced Single Instruction Multiple Data (SIMD) unit, then Arm Compiler for Embedded FuSa can use the vectorizing engine to optimize vectorizable sections of the code. At optimization level `-O1`, you can enable vectorization using `-fvectorize`. At higher optimizations, `-fvectorize` is enabled by default and you can disable it using `-fno-vectorize`. See -fvectorize, -fno-vectorize in the *Arm Compiler for Embedded FuSa Reference Guide* for more information. When using `-fvectorize` with `-O1`, vectorization might be inhibited in the absence of other optimizations which might be present at `-O2` or higher.

As an implementation becomes more complicated, the likelihood that the compiler can auto-vectorize the code decreases. For example, loops with the following characteristics are particularly difficult, or impossible, to vectorize:

- Loops with interdependencies between different loop iterations.
- Loops with break clauses.
- Loops with complex conditions.

The following examples show a loop that Advanced SIMD can vectorize, and a loop that cannot be vectorized easily:

Vectorizable by Advanced SIMD

Copy the following into the file `vectorize.c`:

```
typedef struct tBuffer {
    int a;
    int b;
    int c;
} tBuffer;
tBuffer buffer[8];

void DoubleBuffer1 (void)
{
    int i;
    for (i=0; i<8; i++)
    {
        buffer[i].a *= 2;
        buffer[i].b *= 2;
        buffer[i].c *= 2;
    }
}
```

Compile at optimization level `O2` to enable auto-vectorization:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O2 -S vectorize.c -o
 vectorize.s
```

In `vectorize.s`, the vectorized assembly code contains the Advanced SIMD instructions, for example `vld1`, `vshl`, and `vst1`:

```
DoubleBuffer1:
    ...
    movw    r0, :lower16:buffer
    movt    r0, :upper16:buffer
    vld1.64 {d16, d17}, [r0:128]
    vshl.i32    q8, q8, #1
    vst1.32 {d16, d17}, [r0:128]!
    vld1.64 {d16, d17}, [r0:128]
    vshl.i32    q8, q8, #1
    vst1.32 {d16, d17}, [r0:128]!
    vld1.64 {d16, d17}, [r0:128]
    vshl.i32    q8, q8, #1
    vst1.32 {d16, d17}, [r0:128]!
    vld1.64 {d16, d17}, [r0:128]
    vshl.i32    q8, q8, #1
    vst1.32 {d16, d17}, [r0:128]!
    vld1.64 {d16, d17}, [r0:128]
    vshl.i32    q8, q8, #1
    vst1.32 {d16, d17}, [r0:128]!
    vld1.64 {d16, d17}, [r0:128]
    vshl.i32    q8, q8, #1
    vst1.64 {d16, d17}, [r0:128]
    bx  lr
```

Not vectorizable by Advanced SIMD

Copy the following into the file `nonvectorize.c`:

```
typedef struct tBuffer {
    int a;
    int b;
    int c;
} tBuffer;
tBuffer buffer[8];

void DoubleBuffer2 (void)
{
    int i;
    for (i=0; i<8; i++) {
        buffer[i].a *= 2;
        buffer[i].b *= 2;
        buffer[i].c *= 2;
        if (buffer[i].c > 64)
            break;
    }
}
```

Compile at optimization level `O2` to enable auto-vectorization:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -O2 -S nonvectorize.c -o
 nonvectorize.s
```

`nonvectorize.s` shows that the Advanced SIMD instructions are not generated when compiling the example with the non-vectorizable loop:

```
DoubleBuffer2:
    ...
    movw    r12, :lower16:buffer
    movt    r12, :upper16:buffer
    ldm r12, {r1, r2, r3}
    lsl r0, r3, #1
    cmp r3, #32
    lsl r2, r2, #1
    str r0, [r12, #8]
    lsl r1, r1, #1
    stm r12, {r1, r2}
    bgt .LBB0_8
@ %bb.1:
    add r2, r12, #12
    ldm r2, {r0, r1, r2}
    lsl r3, r2, #1
    lsl r1, r1, #1
    cmp r2, #32
    str r1, [r12, #16]
    lsl r0, r0, #1
    str r3, [r12, #20]
    str r0, [r12, #12]
    bgt .LBB0_8
...
    add r2, r12, #72
    ldm r2, {r0, r1, r2}
    lsl r3, r2, #1
    lsl r1, r1, #1
    cmp r2, #32
    str r1, [r12, #76]
    lsl r0, r0, #1
    str r3, [r12, #80]
    str r0, [r12, #72]
    bxgt lr
.LBB0_7:
    add r2, r12, #84
    add r3, r12, #84
    ldm r2, {r0, r1, r2}
    lsl r1, r1, #1
    lsl r2, r2, #1
    lsl r0, r0, #1
    stm r3, {r0, r1, r2}
.LBB0_8:
    bx  lr
```

Note

Advanced SIMD, also known as Arm® Neon® technology, is a powerful vectorizing unit on Armv7-A and later Application profile architectures. It enables you to write highly optimized code. You can use intrinsics to directly use the Advanced SIMD capabilities from C or C++ code. The intrinsics and their data types are defined in `arm_neon.h`. For more information on Advanced SIMD, see the Arm C Language Extensions, Cortex-A Series Programmer's Guide, and Arm Neon Programmer's Guide.

Using `-fno-vectorize` does not necessarily prevent the compiler from emitting Advanced SIMD instructions. The compiler or linker might still introduce Advanced SIMD instructions, such as when linking libraries that contain these instructions.

To prevent the compiler from emitting Advanced SIMD instructions for AArch64 targets, specify `+nosimd` using `-march` or `-mcpu`:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a+nosimd -O2 -S file.c -o
 file.s
```

To prevent the compiler from emitting Advanced SIMD instructions for AArch32 targets, set the option `-mfpu` to the correct value that does not include Advanced SIMD. For example, set `-mfpu=fp-armv8`.

```
armclang --target=aarch32-arm-none-eabi -march=armv8-a -mfpu=fp-armv8 -O2 -S file.c
 -o file.s
```

Loop termination in C code

If written without caution, the loop termination condition can cause significant overhead. Where possible:

- Use simple termination conditions.
- Write count-down-to-zero loops and test for equality against zero.
- Use counters of type `unsigned int`.

Following any or all of these guidelines, separately or in combination, is likely to result in better code.

The following sample implementations of a routine to calculate `n!` together show the loop termination overhead. The first implementation calculates `n!` using an incrementing loop, while the second routine calculates `n!` using a decrementing loop.

C code for incrementing loops, `increment.c`

```
int fact1(int n)
{
    int i, fact = 1;
    for (i = 1; i <= n; i++)
        fact *= i;
    return (fact);
}
```

Generate the disassembly using:

```
armclang --target=arm-arm-none-eabi -march=armv7-m -Os -S increment.c
```

The disassembly in `increment.s` for incrementing loops contains:

```
fact1:
    ...
    cmp     r0, #1
    itt     lt
    movlt   r0, #1
    bxlt    lr
.LBB0_1:
    mov     r1, r0
```

```
    movs    r0, #1
    movs    r2, #0
    .p2align        2
.LBB0_2:                                @ =>This Inner Loop Header: Depth=1
    adds    r2, #1
    cmp     r1, r2
    mul     r0, r2, r0
    bne     .LBB0_2
@ %bb.3:
    bx      lr
```

C code for decrementing loops, `decrement.c`

```
int fact2(int n)
{
    unsigned int i, fact = 1;
    for (i = n; i != 0; i--)
        fact *= i;
    return (fact);
}
```

Generate the disassembly using:

```
armclang --target=arm-arm-none-eabi -march=armv7-m -Os -S decrement.c
```

The disassembly in `decrement.s` for decrementing loops contains:

```
fact2:
    ...
    cbz     r0, .LBB0_4
    mov     r1, r0
    movs    r0, #1
    .p2align        2
.LBB0_2:                                @ =>This Inner Loop Header: Depth=1
    muls    r0, r1, r0
    subs    r1, #1
    bne     .LBB0_2
    bx      lr
.LBB0_4:
    movs    r0, #1
    bx      lr
```

Comparing the disassemblies shows that the ADD and CMP instruction pair in the incrementing loop disassembly has been replaced with a single SUBS instruction in the decrementing loop disassembly. Because the SUBS instruction updates the status flags, including the Z flag, there is no requirement for an explicit `CMP r1,r2` instruction.

Also, the variable `n` does not have to be available for the lifetime of the loop, reducing the number of registers that have to be maintained. Having fewer registers to maintain eases register allocation. If the original termination condition involves a function call, each iteration of the loop might call the function, even if the value it returns remains constant. In this case, counting down to zero is even more important. For example:

```
for (...; i < get_limit(); ...);
```

The technique of initializing the loop counter to the number of iterations that are required, and then decrementing down to zero, also applies to while and do statements.

Infinite loops

`armclang` considers infinite loops with no side-effects to be undefined behavior, as stated in the C11 and C++11 standards. In certain situations `armclang` deletes or moves infinite loops that have no side-effects, resulting in a program that eventually terminates, or does not behave as expected.

To ensure that a loop executes for an infinite length of time, we recommend writing infinite loops containing an `__asm volatile` statement. The `volatile` keyword tells the compiler to consider that the loop has potential side effects, and therefore prevents the loop from being removed by optimization. It is also good practice to try and put the processor in a low power state in such a loop, until an event or interrupt occurs. The following example shows an infinite loop that is specified as `volatile`, and includes an instruction to put the processor in a low power state until an event occurs:

```
void infinite_loop(void)
{
    while (1)
    {
        __asm volatile("wfe");
    }
}
```

The `volatile` keyword tells `armclang` not to delete or move the loop. The compiler considers the loop to have side-effects, and so it is not removed during optimization.

The `WFE` (Wait for Event) assembler instruction gives a hint to the processor. Writing your loop this way allows processors that implement the `WFE` instruction to enter a low power state until an event or interrupt occurs, so the loop does not consume power unnecessarily. You could also use `WFI` (Wait for Interrupt) to output code that includes the `WFI` instruction, which allows processors that implement `WFI` to wake on an interrupt signal rather than event signal.

For more details on `WFE` and `WFI`, see the relevant Instruction Set Architecture document for the processor you are compiling for.

Related information

Effect of the volatile keyword on compiler optimization on page 73
-O (armclang)
-S (armclang)
pragma unroll
-fvectorize (armclang)

4.3 Inlining functions

Arm® Compiler for Embedded FuSa automatically inlines functions if it decides that inlining the function gives better performance. This inlining does not significantly increase the code size.

However, you can use compiler hints and options to influence or control whether a function is inlined or not.

Table 4-2: Function inlining

Inlining options, keywords, or attributes	Description
`__inline__`	Specify this keyword on a function definition or declaration as a hint to the compiler to favor inlining of the function. However, for each function call, the compiler still decides whether to inline the function. This keyword is equivalent to `__inline`.
`__attribute__((always_inline))`	Specify this function attribute on a function definition or declaration to tell the compiler to always inline this function, with certain exceptions such as for recursive functions. This attribute overrides the `-fno-inline-functions` option.
`__attribute__((noinline))`	Specify this function attribute on a function definition or declaration to tell the compiler to not inline the function. This attribute is equivalent to `__declspec(noinline)`.
`-fno-inline-functions`	A compiler command-line option. Specify this option to the compiler to disable inlining. This option overrides the `__inline__` hint.


Note

- Arm Compiler for Embedded FuSa only inlines functions within the same compilation unit, unless you use Link-Time Optimization. For more information, see Optimizing across modules with Link-Time Optimization.
- C++ and C99 provide the `inline` language keyword. The effect of this `inline` language keyword is identical to the effect of using the `__inline__` compiler keyword. However, the effect in C99 mode is different from the effect in C++ or other C that does not adhere to the C99 standard. For more information, see Inline functions in the *Arm Compiler for Embedded FuSa Reference Guide*.
- Function inlining normally happens at higher optimization levels, such as `-O2`, except when you specify `__attribute__((always_inline))`.

Examples of function inlining

This example shows the effect of `__attribute__((always_inline))` and `-fno-inline-functions` in C99 mode, which is the default behavior for C files. Copy the following code to `file.c`.

```
int bar(int a)
{
    a=a*(a+1);
    return a;
}

__attribute__((always_inline)) static int row(int a)
{
    a=a*(a+1);
    return a;
}

int foo (int i)
{
    i=bar(i);
    i=i-2;
    i=bar(i);
    i++;
```

```
    i=row(i);
    i++;
    return i;
}
```

In the example code, functions `bar` and `row` are identical but function `row` is always inlined. Use the following compiler commands to compile for `-O2` with `-fno-inline-functions` and without `-fno-inline-functions`:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -S file.c -O2 -o file_no_inline.s
 -fno-inline-functions
```

```
armclang --target=arm-arm-none-eabi -march=armv8-a -S file.c -O2 -o
 file_with_inline.s
```

When compiling with `-fno-inline-functions`, the compiler does not inline the function `bar`. When compiling without `-fno-inline-functions`, the compiler inlines the function `bar`. However, the compiler always inlines the function `row` even though it is identical to function `bar`.

Effect of compiling with -fno-inline-functions

```
foo:
    ...
    .save   {r11, lr}
    push    {r11, lr}
    ...
    bl  bar
    sub r0, r0, #2
    bl  bar
    add r1, r0, #1
    add r0, r0, #2
    mul r0, r0, r1
    add r0, r0, #1
    pop {r11, pc}
    ...
```

Effect of compiling without -fno-inline-functions

```
foo:
    ...
    add r1, r0, #1
    mul r0, r1, r0
    sub r1, r0, #2
    sub r0, r0, #1
    mul r0, r0, r1
    add r1, r0, #1
    add r0, r0, #2
    mul r0, r0, r1
    add r0, r0, #1
    bx  lr
    ...
```

Related information

-fno-inline-functions (armclang)
__inline keyword
__attribute__((always_inline)) function attribute

__attribute__((no_inline)) function attribute

4.4 Stack use in C and C++

C and C++ both use the stack intensively.

For example, the stack holds:

- The return address of functions.
- Registers that must be preserved, as determined by the *Procedure Call Standard for the Arm Architecture* (AAPCS) or the *Procedure Call Standard for the Arm 64-bit Architecture* (AAPCS64). For example, when register contents are saved on entry into subroutines.
- Local variables, including local arrays, structures, and unions.
- Classes in C++.

Some stack usage is not obvious, such as:

- If local integer or floating-point variables are spilled (that is, not allocated to a register), they are allocated stack memory.
- Structures are normally allocated to the stack. A space equivalent to `sizeof(struct)` padded to a multiple of `<n>` bytes is reserved on the stack, where `<n>` is `16` for AArch64 state, or `8` for AArch32 state. However, the compiler might try to allocate structures to registers instead.
- If the size of an array is known at compile time, the compiler allocates memory on the stack. Again, a space equivalent to `sizeof(array)` padded to a multiple of `<n>` bytes is reserved on the stack, where `<n>` is `16` for AArch64 state, or `8` for AArch32 state.


Note

Memory for variable length arrays is allocated at runtime, on the heap.

- Several optimizations can introduce new temporary variables to hold intermediate results. The optimizations include CSE elimination, live range splitting, and structure splitting. The compiler tries to allocate these temporary variables to registers. If not, it spills them to the stack. For more information about what these optimizations do, see Overview of optimizations.
- Generally, code that is compiled for processors that only support 16-bit encoded T32 instructions makes more use of the stack than A64 code, A32 code, and code that is compiled for processors that support 32-bit encoded T32 instructions. This is because 16-bit encoded T32 instructions have only eight registers available for allocation, compared to fourteen for A32 code and 32-bit encoded T32 instructions.
- The AAPCS and AAPCS64 require that some function arguments are passed through the stack instead of the registers, depending on their type, size, and order.

Processors for embedded applications have limited memory and therefore the amount of space available on the stack is also limited. You can use Arm® Compiler for Embedded FuSa to determine how much stack space is used by the functions in your application code. The amount of stack that

a function uses depends on factors such as the number and type of arguments to the function, local variables in the function, and the optimizations that the compiler performs.

Methods of estimating stack usage

Stack use is difficult to estimate because it is code dependent, and can vary between runs depending on the code path that the program takes on execution. However, it is possible to manually estimate the extent of stack utilization using the following methods:

- Compile with `-g` and link with `--callgraph` to produce a static callgraph. This callgraph shows information on all functions, including stack usage.
- Link with `--info=stack` or `--info=summarystack` to list the stack usage of all global symbols.
- Use a debugger to set a watchpoint on the last available location in the stack and see if the watchpoint is ever hit. Compile with the `-g` option to generate the necessary DWARF information.
- Use a debugger, and:
  1. Allocate space in memory for the stack that is much larger than you expect to require.
  2. Fill the stack space with copies of a known value, for example, `0xDEADDEAD`.
  3. Run your application, or a fixed portion of it. Aim to use as much of the stack space as possible in the test run. For example, try to execute the most deeply nested function calls and the worst case path that the static analysis finds. Try to generate interrupts where appropriate, so that they are included in the stack trace.
  4. After your application has finished executing, examine the stack space of memory to see how many of the known values have been overwritten. The space has garbage in the used part and the known values in the remainder.
  5. Count the number of garbage values and multiply by `sizeof(value)`, to give their size, in bytes.

  The result of the calculation shows how the size of the stack has grown, in bytes.
- Use a Fixed Virtual Platform (FVP) that corresponds to the target processor or architecture. With a map file, define a region of memory directly below your stack where access is forbidden. If the stack overflows into the forbidden region, a data abort occurs, which a debugger can trap.

Examining stack usage

It is good practice to examine the amount of stack that the functions in your application use. You can then consider rewriting your code to reduce stack usage.

To examine the stack usage in your application, use the linker option `--info=stack`. The following example code shows functions with different numbers of arguments:

```
__attribute__((noinline)) int fact(int n)
{
  int f = 1;
  while (n>0)
  {
    f *= n--;
  }
  return f;
}
```

```
int foo (int n)
{
  return fact(n);
}

int foo_mor (int a, int b, int c, int d)
{
 return fact(a);
}

int main (void)
{
  return foo(10) + foo_mor(10,11,12,13);
}
```

Copy the code example to `file.c` and compile it using the following command:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c -g file.c -o file.o
```

Compiling with the `-g` option generates the DWARF frame information that `armlink` requires for estimating the stack use. Run `armlink` on the object file using `--info=stack`:

```
armlink file.o --info=stack
```

For the example code, `armlink` shows the amount of stack that the various functions use. Function `foo_mor` has more arguments than function `foo`, and therefore uses more stack.

```
Stack Usage for fact 0xc bytes.
Stack Usage for foo 0x8 bytes.
Stack Usage for foo_mor 0x10 bytes.
Stack Usage for main 0x8 bytes.
```

You can also examine stack usage using the linker option `--callgraph`:

```
armlink file.o --callgraph -o FileImage.axf
```

This command outputs a file called `FileImage.htm` which contains the stack usage information for the various functions in the application.

```
fact (ARM, 84 bytes, Stack size 12 bytes, file.o(.text))

[Stack]

Max Depth = 12
Call Chain = fact

[Called By]
>>   foo_mor
>>   foo
foo (ARM, 36 bytes, Stack size 8 bytes, file.o(.text))

[Stack]

Max Depth = 20
Call Chain = foo >> fact
```

```
[Calls]
>>   fact

[Called By]
>>   main
foo_mor (ARM, 76 bytes, Stack size 16 bytes, file.o(.text))

[Stack]

Max Depth = 28
Call Chain = foo_mor >> fact

[Calls]
>>   fact

[Called By]
>>   main
main (ARM, 76 bytes, Stack size 8 bytes, file.o(.text))

[Stack]

Max Depth = 36
Call Chain = main >> foo_mor >> fact

[Calls]
>>   foo_mor
>>   foo

[Called By]
>>   __rt_entry_main (via BLX)
```

See --info and --callgraph for more information on these options.

Methods of reducing stack usage

In general, you can lower the stack requirements of your program by:

- Writing small functions that only require a few variables.
- Avoiding the use of large local structures or arrays.
- Avoiding recursion.
- Minimizing the number of variables that are in use at any given time at each point in a function.
- Using C block scope syntax and declaring variables only where they are required, so that distinct scopes can use the same memory.

4.5 Packing data structures

You can reduce the amount of memory that your application requires by packing data into structures. This is especially important if you need to store and access large arrays of data in embedded systems.

If individual data members in a structure are not packed, the compiler can add padding within the structure for faster access to individual members, based on the natural alignment of each member. Arm® Compiler for Embedded FuSa provides a pragma and attribute to pack the members in a structure or union without any padding.

Table 4-3: Packing members in a structure or union

Pragma or attribute	Description
`#pragma pack (<n>)`	For each member, if <n> bytes is less than the natural alignment of the member, then set the alignment to <n> bytes, otherwise the alignment is the natural alignment of the member. For more information see #pragma pack(n) and __alignof__.
`__attribute__((packed))`	This is equivalent to `#pragma pack(1)`. However, the attribute can also be used on individual members in a structure or union.

Packing the entire structure

To pack the entire structure or union, use `__attribute__((packed))` or `#pragma pack(n)` to the declaration of the structure as shown in the code examples. The attribute and pragma apply to all the members of the structure or union. If the member is a structure, then the structure has an alignment of 1-byte, but the members of that structure continue to have their natural alignment.

When using `#pragma pack(n)`, the alignment of the structure is the alignment of the largest member after applying `#pragma pack(n)` to the structure.

Each example declares two objects `c` and `d`. Copy each example into `file.c` and compile:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c file.c -o file.o
```

For each example use linker option `--info=sizes` to examine the memory used in `file.o`.

```
armlink file.o --info=sizes
```

The linker output shows the total memory used by the two objects `c` and `d`. For example:

```
Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

  36          0          0          0         24          0   str.o
----------------------------------------------------------------------
  36          0         16          0         24          0   Object Totals
```

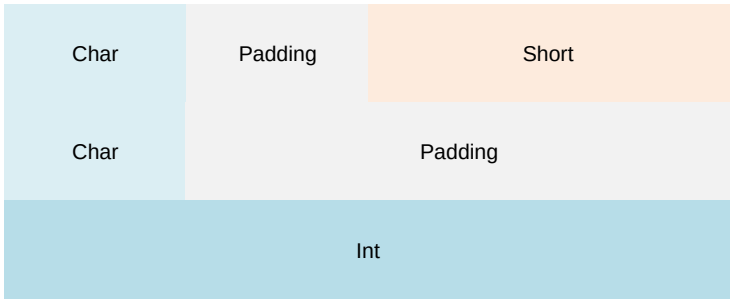
Packing a 12-byte structure using natural alignment

The alignment of the structure is the natural alignment of the largest member. In this example, the largest member is an `int`.

```
struct stc
{
    char one;
    short two;
    char three;
    int four;
} c,d;


int main (void)
{
    c.one=1;
    return 0;
}
```

Figure 4-1: Structure without packing attribute or pragma



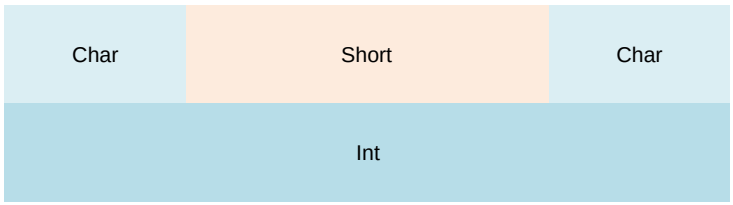
Packing an 8-byte structure using __attribute__((packed)) type attribute

The alignment of the structure is 1 byte:

```
struct __attribute__((packed)) stc
{
    char one;
    short two;
    char three;
    int four;
} c,d;


int main (void)
{
    c.one=1;
    return 0;
}
```

Figure 4-2: Structure with attribute packed



Packing an 8-byte structure using #pragma pack

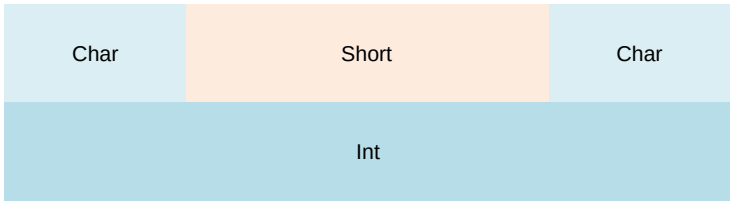
The alignment of the structure is 1 byte:

```
#pragma pack (1)
struct stc
{
    char one;
    short two;
    char three;
```

```
    int four;
} c,d;


int main (void)
{
    c.one=1;
    return 0;
}
```

Figure 4-3: Structure with pragma pack with 1 byte alignment



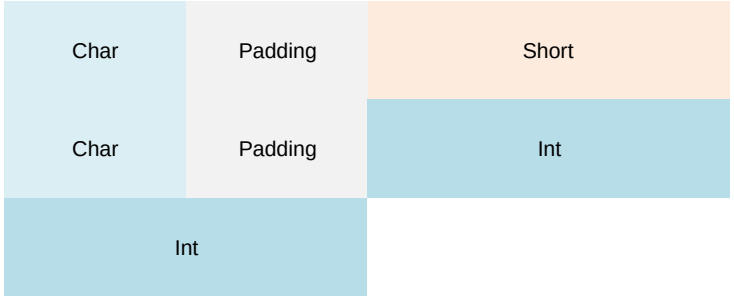
Packing a 10-byte structure using #pragma pack

The alignment of the structure is 2 bytes:

```
#pragma pack (2)
struct stc
{
    char one;
    short two;
    char three;
    int four;
} c,d;


int main (void)
{
    c.one=1;
    return 0;
}
```

Figure 4-4: Structure with pragma pack with 2 byte alignment



Packing a 12-byte structure using #pragma pack

The alignment of the structure is 4 bytes:

```
#pragma pack (4)
struct stc
{
    char one;
    short two;
    char three;
    int four;
} c,d;


int main (void)
{
    c.one=1;
    return 0;
}
```

Figure 4-5: Structure with pragma pack with 4 byte alignment

Packing individual members in a structure

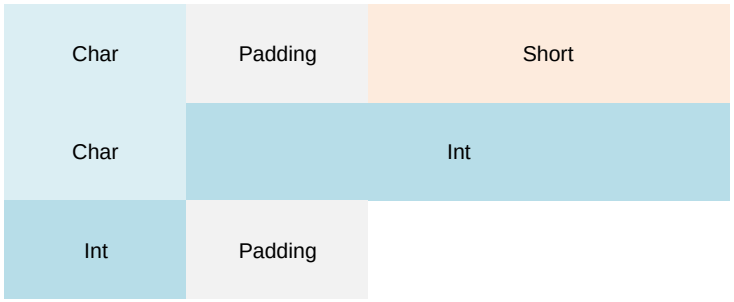
To pack individual members of a structure, use `__attribute__((packed))` on the member. This aligns the member to a byte boundary and therefore reduces the amount of memory required by the structure as a whole. It does not affect the alignment of the other members. Therefore the alignment of the whole structure is equal to the alignment of the largest member without the `__attribute__((packed))`.

The alignment of the 10-byte structure is 2 bytes because the largest member without `__attribute__((packed))` is `short`:

```
struct stc
{
    char one;
    short two;
    char three;
    int  __attribute__((packed)) four;
} c,d;


int main (void)
{
    c.one=1;
    return 0;
}
```

Figure 4-6: Structure with attribute packed on individual member



Accessing packed members from a structure

If a member of a structure or union is packed and therefore does not have its natural alignment, then to access this member, you must use the structure or union that contains this member. You must not take the address of such a packed member to use as a pointer, because the pointer might be unaligned. Dereferencing such a pointer can be unsafe even when unaligned accesses are supported by the target, because certain instructions always require word-aligned addresses.

Note

If you take the address of a packed member, in most cases, the compiler generates a warning.

```
struct __attribute__((packed)) bar
{
  char x;
  short y;
};

short get_y(struct bar *s)
{
    // Correct usage: the compiler does not use unaligned accesses
    // unless they are allowed.
    return s->y;
}

short get2_y(struct bar *s)
{
    short *p = &s->y; // Incorrect usage: 'p' might be an unaligned pointer.
    return *p;        // This code might cause an unaligned access.
}
```

Related information

pragma pack
__attribute__((packed)) type attribute
__attribute__((packed)) variable attribute

4.6 Optimizing for code size or performance

The compiler and associated tools use many techniques for optimizing your code. Some of these techniques improve the performance of your code, while other techniques reduce the size of your code.

Different optimizations often work against each other. That is, techniques for improving code performance might result in increased code size, and techniques for reducing code size might reduce performance. For example, the compiler can unroll small loops for higher performance, with the disadvantage of increased code size.

The default optimization level is `-O0`. At `-O0`, `armclang` does not perform optimization.

The following `armclang` options help you optimize for code performance:

`-O1`, `-O2`, or `-O3`
: Specify the level of optimization to be used when compiling source files. A higher number implies a higher level of optimization for performance.

`-Ofast`
: Enables all the optimizations from `-O3` together with other aggressive optimizations that might violate strict compliance with language standards.

`-Omax`
: Enables all the optimizations from `-Ofast` together with Link-Time Optimization (LTO).

The following `armclang` options help you optimize for code size:

`-Os`
: Performs optimizations to reduce the code size at the expense of a possible increase in execution time. This option aims for a balanced code size reduction and fast performance.

`-Oz`
: Optimizes for smaller code size.

`-Omin`
: Minimum image size. Specifically targets minimizing code size. Enables all the optimizations from level -Oz, together with:

- LTO aimed at removing unused code and data, while also trying to optimize global memory accesses.
- Virtual function elimination, which is a particular benefit to C++ users.

For more information on optimization levels, see Selecting optimization options.


Note

You can also set the optimization level for the linker with the `armlink` option `--lto_level`. The optimization levels available for `armlink` are the same as the `armclang` optimization levels.

-fshort-enums

Allows the compiler to set the size of an enumeration type to the smallest data type that can hold all enumerator values.

-fshort-wchar

Sets the size of `wchar_t` to 2 bytes.

-fno-exceptions

C++ only. Disables the generation of code that is required to support C++ exceptions.

-fno-rtti

C++ only. Disables the generation of code that is required to support Run-Time Type Information (RTTI) features.

-mthumb

In AArch32 state, A- and R-profile processors support both the A32 instruction set (formerly ARM), and the T32 instruction set (formerly Thumb®).

T32 offers significant code size improvements compared to A32, with comparable performance. Therefore, if you are compiling for AArch32 state for a target that supports both A32 and T32 instructions, consider compiling with `-mthumb` to reduce the size of your code.

The following `armclang` option helps you optimize for both code size and code performance:

-flto

Enables LTO, which enables the linker to make additional optimizations across multiple source files. See Optimizing across modules with Link-Time Optimization for more information.


Note

If you want to use LTO when invoking `armlink` separately, you can use the `armlink` option `--lto_level` to select the LTO optimization level that matches your optimization goal.

Also, choices you make during coding can affect optimization. For example:

- Optimizing loop termination conditions can improve both code size and performance. In particular, loops with counters that decrement to zero usually produce smaller, faster code than loops with incrementing counters.
- Manually unrolling loops by reducing the number of loop iterations, but increasing the amount of work that is done in each iteration, can improve performance at the expense of code size.
- Reducing debug information in objects and libraries reduces the size of your image.
- Using inline functions offers a trade-off between code size and performance.
- Using intrinsics can improve performance.

4.7 Methods of minimizing function parameter passing overhead

There are several ways in which you can minimize the overhead of passing parameters to functions.

For example:

- In AArch64 state, 8 integer and 8 floating-point arguments (16 in total) can be passed efficiently. In AArch32 state, ensure that functions take four or fewer arguments if each argument is a word or less in size.
- In C++, ensure that nonstatic member functions take fewer arguments than the efficient limit, because in AArch32 state the implicit `this` pointer argument is usually passed in `R0`.
- Ensure that a function does a significant amount of work if it requires more than the efficient limit of arguments. The work that the function does then outweighs the cost of passing the stacked arguments.
- Put related arguments in a structure, and pass a pointer to the structure in any function call. Pointing to a structure reduces the number of parameters and increases readability.
- For AArch32 state, minimize the number of `long long` parameters, because these use two argument registers that have to be aligned on an even register index.
- For AArch32 state, minimize the number of `double` parameters when using software floating-point.

4.8 Optimizing across modules with Link-Time Optimization

At link time, more optimization opportunities are available because source code from different modules can be optimized together.

By default, the compiler optimizes each source module independently, translating C or C++ source code into an ELF file containing object code. At link time, the linker combines all the ELF object files into an executable by resolving symbol references and relocations. Compiling each source file separately means that the compiler might miss some optimization opportunities, such as cross-module inlining.

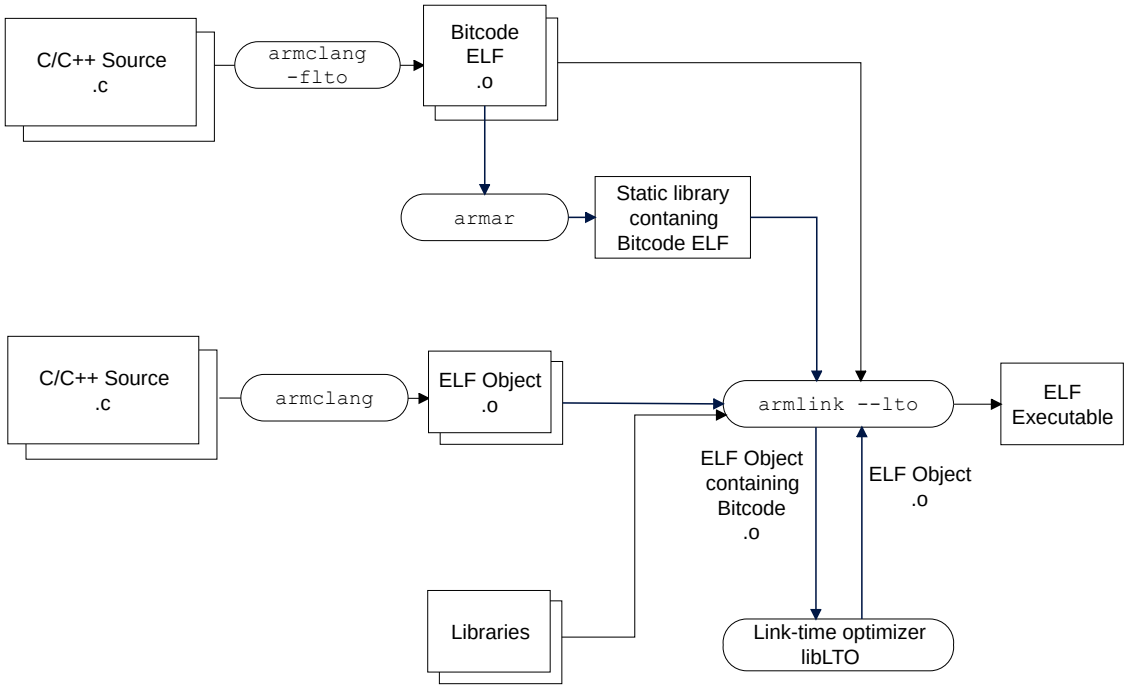
When Link-Time Optimization (LTO) is enabled, the compiler translates source code into an intermediate form called LLVM bitcode. At link time, the linker collects all files containing bitcode together and sends them to the link-time optimizer, `libLTO`. `libLTO` is provided as a library:

- `libLTO.so` on Linux.
- `LTO.dll` on Windows.

Collecting modules together means that the link-time optimizer can perform more optimizations because it has more information about the dependencies between modules. The link-time

optimizer then sends a single ELF object file back to the linker. Finally, the linker combines all object and library code to create an executable.

Figure 4-7: Link-Time Optimization



Note

In this figure, ELF Object containing Bitcode is an ELF file that does not contain normal code and data. Instead, it contains a section that is called `.llvm.lto` that holds LLVM bitcode. In Arm® Compiler for Embedded FuSa versions earlier than 6.21, the section is called `.llvmbc`.

Sections `.llvm.lto` and `.llvmbc` are reserved. You must not create a `.llvm.lto` or `.llvmbc` section with `__attribute__((section("<name>")))`, for example, `__attribute__((section(".llvmbc")))`.

Caution

LTO performs aggressive optimizations by analyzing the dependencies between bitcode format objects. Such aggressive optimizations can result in the removal of unused variables and functions in the source code.

4.8.1 Enabling Link-Time Optimization

You must enable Link-Time Optimization (LTO) in both `armclang` and `armlink`.

Procedure

1. At compilation time, use the `armclang` option `-flto` to produce ELF files suitable for LTO. These ELF files contain bitcode in a `.llvm.lto` section.


Note

The `armclang` options `-Omax` and `-Omin` automatically enable the `-flto` option.

2. At link time, use the `armlink` option `--lto` to enable LTO for the specified bitcode files.


Note

If you use the `-flto` option without the `-c` option, `armclang` automatically passes the `--lto` option to `armlink`.

Example 4-1: Link-Time Optimization

The examples described in Link-Time Optimization examples show how to perform LTO across all source files, or a subset of source files.

4.8.2 Restrictions with Link-Time Optimization

Link-Time Optimization (LTO) has a few restrictions in Arm® Compiler for Embedded FuSa 6. Future releases might have fewer restrictions and more features. The user interface to LTO might change in future releases.

Partial linking

The `armlink` option `--partial` only works with ELF files. If the linker detects a file containing bitcode, it gives an error message.

Scatter-loading

The output of the link-time optimizer is a single ELF object file that by default is given a temporary filename. This ELF object file contains sections and symbols just like any other ELF object file, and Input section selectors match the sections and symbols as normal.

Use the `armlink` option `--lto_intermediate_filename` to name the ELF object file output. You can reference this ELF file name in the scatter file.

We recommend that LTO is only performed on code and data that does not require precise placement in the scatter file. That is, placement with general Input section selectors such as `*(+RO)` and `.ANY(+RO)` used to select sections that LTO generates. See Scatter file section or

object placement with Link-Time Optimization for an example of building an image using LTO and with a scatter file to place named sections.

It is not possible to match bitcode in `.llvm.lto` sections by name in a scatter file.


Note

The scatter-loading interface is subject to change in future versions of Arm Compiler for Embedded FuSa 6.

Executable and library compatibility

The `armclang` executable and the `libLTO` library must come from:

- The same Arm Compiler for Embedded FuSa 6 installation.
- The same version of the compiler.

Any use of `libLTO` other than the library supplied with Arm Compiler for Embedded FuSa 6 is unsupported.

Other restrictions

- You cannot currently use LTO for building ROPI/RWPI images.
- Object files that LTO produces contain build attributes that are the default for the target architecture. If you use the `armlink` options `--cpu` or `--fpu` when LTO is enabled, `armlink` can incorrectly report that the attributes in the file that the link-time optimizer produces are incompatible with the provided attributes.

  
  Note

  Build attribute compatibility checking is supported only for AArch32 state.

- LTO does not honor `armclang` options `-fno-function-sections` and `-fno-data-sections`. The output of the LTO code generator is the equivalent of the `armclang` options `-ffunction-sections` and `-fdata-sections`.
- LTO does not honor the `armclang` option `-mexecute-only`. If you use the `armclang` options `-flto` or `-Omax`, then the compiler cannot generate execute-only code.
- LTO does not work correctly when two bitcode files are compiled for different targets.
- All bitcode objects and libraries must be compiled and linked with the same version of `armclang`. Therefore, any shared library built using LTO, including any code compiled using the `-Omax` or `-Omin` optimization options, can only be linked with objects using the same compiler version. If you attempt to link objects that were compiled with a different version, and if link-time optimization is used, then an error is generated.
- The linker cannot see references to symbols from inline assembly in bitcode files. If the symbols have not been referenced from elsewhere the linker reports an undefined reference error.

- LTO can interfere with the correct reporting of errors when using file-scope inline assembly. For more information, see File-scope inline assembly.

4.8.3 Link-Time Optimization examples

These examples show how to perform Link-Time Optimization (LTO) across all source files, or a subset of source files.

Example: Optimizing all source files

The following example performs LTO across all source files:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -flto src1.c src2.c src3.c -o
 output.axf
```

This example does the following:

1. `armclang` compiles the C source files `src1.c`, `src2.c`, and `src3.c` to the ELF files `src1.o`, `src2.o`, and `src3.o`. These ELF files contain bitcode, and therefore `fromelf` cannot disassemble them.
2. `armclang` automatically invokes `armlink` with the `--lto` option.
3. `armlink` passes the bitcode files `src1.o`, `src2.o`, and `src3.o` to the link-time optimizer to produce a single optimized ELF object file.
4. `armlink` creates the executable `output.axf` from the ELF object file.


Note

In this example, as `armclang` automatically calls `armlink`, the link-time optimizer has the same optimization level as `armclang`. As no optimization level is specified for `armclang`, it is the default optimization level `-O0`, and `--lto_level=O0`.

Example: Optimizing a subset of source files

The following example performs LTO for a subset of source files.

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c src1.c -o src1.o
armclang --target=arm-arm-none-eabi -march=armv8-a -c -flto src2.c -o src2.o
armclang --target=arm-arm-none-eabi -march=armv8-a -c -flto src3.c -o src3.o
armlink --lto src1.o src2.o src3.o -o output.axf
```

This example does the following:

1. `armclang` compiles the C source file `src1.c` to the ELF object file `src1.o`.
2. `armclang` compiles the C source files `src2.c` and `src3.c` to the ELF files `src2.o` and `src3.o`. These ELF files contain bitcode.
3. `armlink` passes the bitcode files `src2.o` and `src3.o` to the link-time optimizer to produce a single optimized ELF object file.
4. `armlink` combines the ELF object file `src1.o` with the object file that the link-time optimizer produces to create the executable `output.axf`.

Note

In this example, because `armclang` and `armlink` are called separately, they have independent optimization levels. As no optimization level is specified for `armclang` or `armlink`, `armclang` has the default optimization level `-O0` and the link-time optimizer has the default optimization level `--lto_level=O2`. You can call `armclang` and `armlink` with any combination of optimization levels.

4.8.4 Removing unused code across multiple object files

Link-Time Optimization (LTO) might remove unused functions and data across multiple object files, particularly when there are no references to those functions and data. However, functions marked as no inline are not removed.

About this task

In this example:

- The function `main()` calls an externally defined function `function()`, and returns the value that `function()` returns. Because this function is externally defined, the compiler cannot inline or otherwise optimize it when compiling `main.c`, without using LTO.
- The file `function.c` contains the following functions:

  `function()`
  : If the parameter `a` is nonzero, `function()` conditionally calls a function `printit()`.

  `printit()`
  : This function prints a message.

In this case, `function()` is called with the parameter `a == 0`, so `printit()` is not called at run time.

Example code that is used in the following procedure:

```
// main.c
extern int function(int a);

int main(void)
{
    return function(0);
}

// functions.c
#include <stdio.h>

int function(int a);
void printit(void);

/* function() conditionally calls printit()
   depending on the value of a
 */
int function(int a)
{
  if (a == 0)
  {
      return 0;
  }
  else
  {
```

```
      printit();
      return 0;
  }
}

void printit(void)
{
  printf("a is non-zero.\n");
}
```

Procedure

1. Build the example code with LTO disabled:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -c main.c -o main.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -c functions.c -o
 functions.o
armlink main.o functions.o -o image_without_lto.axf
fromelf --text -c -z image_without_lto.axf
```

   The compiler cannot inline the call to `function()` because it is in a different object from `main()`. Therefore, the compiler must keep the conditional call to `printit()` within `function()`, because the compiler does not have any information about the value of the parameter `a` while `functions.c` is being compiled:

```
...
$a.0
function
    0x00008bd8:    e3500000    ..P.    CMP      r0,#0
    0x00008bdc:    0a000004    ....    BEQ      0x8c18 ; function + 28
    0x00008be0:    e92d4800    .H-.    PUSH     {r11,lr}
    0x00008be4:    e3080c6a    j...    MOV      r0,#0x8c6a
    0x00008be8:    e3400000    ..@.    MOVT     r0,#0
    0x00008bec:    fafffd1f    ....    BLX      puts ; 0x8094
    0x00008bf0:    e8bd4800    .H..    POP      {r11,lr}
    0x00008bf4:    e3a00000    ....    MOV      r0,#0
    0x00008bf8:    e12fff1e    ../.    BX       lr
main
    0x00008bfc:    e3a00000    ....    MOV      r0,#0
    0x00008c00:    eafffff4    ....    B        function ; 0x8bfc
...
```

   Also, `printit()` uses the Arm C library function `printf()`. In this example, `printf()` is optimized to `puts()` and inlined into `function()`. Therefore, the linker must include the relevant C library code to allow the `puts()` function to be used. Including the C library code results in a large amount of uncalled code being included in the image. The output from the `fromelf` utility shows the resulting overall image size:

```
** Object/Image Component Sizes

Code (inc. data)   RO Data   RW Data   ZI Data   Debug   Object Name

3166        202         46        16       348    1824   image_without_lto.axf
3166        202         46        16         0       0   ROM Totals for
 image_without_lto.axf
```

2. Build the example code with LTO enabled:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -flto -c main.c -o main.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -flto -c functions.c -o
 functions.o
```

```
armlink --lto main.o functions.o -o image_with_lto.axf
fromelf --text -c -z image_with_lto.axf
```

Although the compiler does not have any information about the call to `function()` from `main()` when compiling `functions.c`, at link time, it is known that:

- `function()` is only ever called once, with the parameter `a == 0`.
- `printit()` is never called.
- The Arm C library function `puts()` is never called.

Because LTO is enabled, this extra information is used to make the following optimizations:

- Inlining the call to `function()` into `main()`.
- Removing the code to conditionally call `printit()` from `function()` entirely.
- Removing the C library code that allows use of the `puts()` function.

```
...
$a.0
main
    0x00008128:    e3a00000    ....    MOV      r0,#0
    0x0000812c:    e12fff1e    ../.    BX       lr
...
```

Also, this optimization means that the overall image size is much lower. The output from the `fromelf` utility shows the reduced image size:

```
** Object/Image Component Sizes

Code (inc. data)   RO Data   RW Data   ZI Data   Debug   Object Name

332         24         16         0         96     504   image_with_lto.axf
332         24         16         0          0       0   ROM Totals for
 image_with_lto.axf
```

Related information

Optimizing for code size or performance on page 95
Optimizing across modules with Link-Time Optimization on page 98
How optimization affects the debug experience on page 112
-O (armclang)

4.9 Scatter file section or object placement with Link-Time Optimization

Turning on Link-Time Optimization (LTO) using either `-Omax` or `-flto` means that at link time, all object files are merged into one. If a project is using a scatter file that places sections or objects in

specific regions, both the scatter file and the project source code must be modified to ensure the placement works with LTO.

In general:

- Scatter files with object names that are used in input selection patterns, such as `foo.o(+RO)` do not work with LTO.
- Scatter files with section names that are used in input selection patterns, where the section name corresponds to an inlined function, do not work.

In such circumstances, the linker might report a warning such as:

```
L6314W: No section matches pattern <module>(<section>).
```

To use scatter file section or object placement with LTO, the following changes must be made to a project:

- Compile all source files that are built with LTO enabled with `-fno-inline-functions`.
- Modify each source file that is built with LTO enabled to use `#pragma clang section` to place all functions in that source file into sections with a name unique to that source file.
- Modify the scatter file to use section names instead of object file names.

Example code

The following example code is used in the example sections, unless specified otherwise. In this code, all functions in `foo.c` must be placed in an execution region `EXEC_FOO`, and all functions in `bar.c` must be placed in an execution region `EXEC_BAR`:

`variables.c`:

```
const int foo_int = 42;

const int bar_int = 42;
```

`foo.c`:

```
#include <stdio.h>

extern const int foo_int;

void foo(void)
{
    printf("The answer from foo is: %d\n", foo_int);
}
```

`bar.c`:

```
#include <stdio.h>

extern const int bar_int;

void bar(void)
{
    printf("The answer from bar is: %d\n", bar_int);
}
```

main.c:

```
extern void foo(void);
extern void bar(void);

int main(void)
{
    foo();
    bar();

    return 0;
}
```

scatter.scat:

```
LOAD 0x0
{
    EXEC_ANY +0x0
    {
        .ANY(+RO, +RW, +ZI)
    }

    EXEC_FOO +0x0 ALIGN 1024
    {
        foo.o(+RO)
    }

    EXEC_BAR +0x0 ALIGN 1024
    {
        bar.o(+RO)
    }

    ARM_LIB_STACKHEAP +0x0 ALIGN 8 EMPTY 4096 {}
}
```

Example: Building without LTO enabled

Build the example code with:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -c variables.c -o variables.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -c foo.c -o foo.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -c bar.c -o bar.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -c main.c -o main.o
armlink --scatter=scatter.scat variables.o foo.o bar.o main.o -o image.axf --map --
list=image.lst
```

The memory map from the listing file `image.lst` shows that `EXEC_FOO` and `EXEC_BAR` contain code from `foo.c` and `bar.c` respectively, as intended:

```
Execution Region EXEC_FOO (Base: 0x00001000, Size: 0x00000028, Max: 0xffffffff,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00001000   0x00000028   Code   RO            6    .text.foo           foo.o


Execution Region EXEC_BAR (Base: 0x00001400, Size: 0x00000018, Max: 0xffffffff,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00001400   0x00000018   Code   RO           10    .text.bar           bar.o
```

Example: Building with LTO enabled

Build the example code with LTO enabled:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -c variables.c -o variables.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -flto -c foo.c -o foo.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -flto -c bar.c -o bar.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -flto -c main.c -o main.o
armlink --scatter=scatter.scat variables.o foo.o bar.o main.o -o image.axf --lto --
map --list=image.lst
```

In this example, compiling `variables.c` without `-flto` has no effect on the result of running the image. However, compiling the file without `-flto` is required when placing data with named sections.

The linker reports:

```
"scatter.scat", line 10 (column 16): Warning: L6314W: No section matches pattern
 foo.o(RO).
"scatter.scat", line 15 (column 16): Warning: L6314W: No section matches pattern
 bar.o(RO).
Finished: 0 information, 2 warning and 0 error messages
```

Also, the memory map from the listing file `image.lst` shows that `EXEC_FOO` and `EXEC_BAR` are empty:

```
Execution Region EXEC_FOO (Base: 0x00001000, Size: 0x00000000, Max: 0xffffffff,
 ABSOLUTE)

**** No section assigned to this execution region ****


Execution Region EXEC_BAR (Base: 0x00001000, Size: 0x00000000, Max: 0xffffffff,
 ABSOLUTE)

**** No section assigned to this execution region ****
```

These execution regions are empty because LTO has inlined all functions within `foo.c` and `bar.c`. Therefore, the functions are no longer available for placement with a scatter file.

Example: Building with LTO enabled and function inlining disabled

Next, try disabling function inlining using `-fno-inline-functions`. Build the example code with:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -c variables.c -o variables.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -flto -fno-inline-functions -
c foo.c -o foo.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -flto -fno-inline-functions -
c bar.c -o bar.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -flto -fno-inline-functions -
c main.c -o main.o
armlink --scatter=scatter.scat variables.o foo.o bar.o main.o -o image.axf --lto --
map --list=image.lst
```

In this example, compiling `variables.c` without `-flto` has no effect on the result of running the image. However, compiling the file without `-flto` is required when placing data with named sections.

The linker still reports:

```
"scatter.scat", line 10 (column 16): Warning: L6314W: No section matches pattern
 foo.o(RO).
"scatter.scat", line 15 (column 16): Warning: L6314W: No section matches pattern
 bar.o(RO).
Finished: 0 information, 2 warning and 0 error messages.
```

The reason is that, even though function inlining is disabled, all code from `main.c`, `foo.c`, and `bar.c` is part of the same intermediate LTO object file. Therefore, at the final link stage within the LTO process, `foo.o` and `bar.o` do not exist as separate object files.

The memory map in the listing file `image.lst` shows that the code from `foo.c` and `bar.c` is now placed in the `EXEC_ANY` execution region instead:

```
Execution Region EXEC_ANY (Base: 0x00000000, Size: 0x00000f94, Max: 0xffffffff,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object
...
0x00000d90   0x00000010   Code   RO          441    .text.bar           lto-
llvm-68b687.o
0x00000da0   0x00000020   Code   RO          439    .text.foo           lto-
llvm-68b687.o
0x00000dc0   0x00000014   Code   RO          443    .text.main          lto-
llvm-68b687.o
...
```

In this example, `lto-llvm-68b687.o` is the LTO intermediate filename that the linker generates. However, this filename might be different when linking again.

Although you can change the LTO intermediate name using the `armlink` command-line option `--lto_intermediate_filename`, it does not help in this use case. Instead, you must use section names.

Example: Using section names for functions and data within a C language source file

The easiest way to specify section names for all functions or data within a C language source file is to use `#pragma clang section`. Alternatively, you can use `__attribute__((section("<section>")))` for specific functions and data.

For this example, rewrite the example code in the files `variables.c`, `foo.c`, and `bar.c` as follows:

`variables.c`:

```
const int __attribute__((section("foo_rodata"))) foo_int = 42;

const int __attribute__((section("bar_rodata"))) bar_int = 42;
```

`foo.c`:

```
#include <stdio.h>

extern const int foo_int;

#pragma clang section text="foo_rotext"

void foo(void)
```

```
{
    printf("The answer is: %d", foo_int);
}
```

bar.c:

```
#include <stdio.h>

extern const int bar_int;

#pragma clang section text="bar_rotext"

void bar(void)
{
    printf("The answer is: %d", bar_int);
}
```

`#pragma clang section text="foo_rotext"` specifies that code in `foo.c` is placed in the named section `foo_rotext` for the code that is generated.

The `__attribute__((section("foo_rodata")))` variable attribute specifies that `foo_int` in `variables.c` is to be placed in the named section `foo_rodata` for the read-only data that is generated.

Similar names are specified in `bar.c` and `variables.c` for the code and data generated by that file. You can rewrite `scatter.scat` to place these section names as follows:

scatter.scat:

```
LOAD 0x0
{
    EXEC_ANY +0x0
    {
      .ANY(+RO, +RW, +ZI)
    }

    EXEC_FOO +0x0 ALIGN 1024
    {
      *(foo_rotext)
      *(foo_rodata)
    }

    EXEC_BAR +0x0 ALIGN 1024
    {
      *(bar_rotext)
      *(bar_rodata)
    }

    ARM_LIB_STACKHEAP +0x0 ALIGN 8 EMPTY 4096 {}
}
```

Example: Building with LTO enabled, function inlining disabled, and using section names instead of object file names

Build the modified example with:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -c variables.c -o variables.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -flto -fno-inline-functions -
c foo.c -o foo.o
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -flto -fno-inline-functions -
c bar.c -o bar.o
```

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -flto -fno-inline-functions -
c main.c -o main.o
armlink --scatter=scatter.scat variables.o foo.o bar.o main.o -o image.axf --lto --
map --list=image.lst
```

Because we are placing the data in named sections with a scatter file, and that data is in a separate file from the code, then we have to build the `variables.c` file without `-flto`. See *Scatter-loading* in Restrictions with Link-Time Optimization for more information.

The linker does not report any warnings. Also, the memory map from the listing file `image.lst` shows that `EXEC_FOO` and `EXEC_BAR` contain the code from the expected sections:

```
Execution Region EXEC_FOO (Base: 0x00001000, Size: 0x0000002c, Max: 0xffffffff,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00001000   0x00000028   Code   RO          442    foo_rotext          lto-
llvm-5660c0.o
0x00001028   0x00000004   Data   RO            3    foo_rodata          variables.o


Execution Region EXEC_BAR (Base: 0x00001400, Size: 0x0000001c, Max: 0xffffffff,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00001400   0x00000018   Code   RO          444    bar_rotext          lto-
llvm-5660c0.o
0x00001418   0x00000004   Data   RO            4    bar_rodata          variables.o
```

The key difference between this LTO approach and the non-LTO approach with object file names is that in this approach, the function names are not visible in the listing file. To verify that the sections `foo_rotext` and `bar_rotext` contain the functions from `foo.c` and `bar.c` respectively, examine the symbol table from the `fromelf --text -s` output:

```
fromelf --text -s image.axf -o image.txt

...
** Section #8 '.symtab' (SHT_SYMTAB)
    Size   : 7328 bytes (alignment 4)
    String table #9 '.strtab'
    Last local symbol no. 309

    Symbol table .symtab (457 symbols, 309 local)

      #  Symbol Name                 Value      Bind  Sec  Type  Vis  Size
    ============================================================================
    ...
    304  foo                         0x00001000   Lc    4  Code  De   0x14
    305  bar                         0x00001400   Lc    5  Code  De   0x14
    ...
    454  foo_int                     0x00001028   Gb    4  Data  Hi   0x4
    455  bar_int                     0x00001418   Gb    5  Data  Hi   0x4
```

The addresses for these functions in the output from the `fromelf` utility correspond to the execution region addresses in the memory map from the listing file `image.lst`. The symbol table also confirms the location of the `int` constants sections `foo_rodata` and `bar_rodata`.

Other considerations

Other approaches you might want to consider:

- If you plan to build a project with LTO eventually, it might be better to use section names instead of object file names within scatter files using the method shown in this example. This approach is compatible both with and without LTO.
- If you disable LTO, it is better to also remove `-fno-inline-functions`, because doing so allows the compiler to perform inlining optimizations.
- If disabling function inlining entirely is not required, then use the attribute `__attribute__((noinline))` on each function that is not to be inlined. This approach can help achieve a better balance between explicit code placement and cross-file function inlining optimizations.

Related information

Optimizing across modules with Link-Time Optimization on page 98
-fno-inline-functions (armclang)
-flto (armclang)
-O (armclang)
__attribute__((noinline)) function attribute
__attribute__((section("name"))) function attribute
__attribute__((section("name"))) variable attribute
#pragma clang section
--lto (armlink)
--lto_intermediate_filename (armlink)
Scatter-loading Features
Scatter File Syntax

4.10 How optimization affects the debug experience

Higher optimization levels result in an increasingly degraded debug view because the mapping of object code to source code is not always clear. The compiler might perform optimizations that debug information cannot describe.

Therefore, there is a trade-off between optimizing code and the debug experience.

For good debug experience, we recommend `-O1` rather than `-O0`. When using `-O1`, the compiler performs certain optimizations, but the structure of the generated code is still close to the source code.

For more information, see Selecting optimization options.

4.11 Literal pool options in armclang

`armclang` does not provide explicit controls for generating literal pools. Instead, `armclang` provides a mechanism that lets it share literals between functions that are not in the same section. `armclang` marks the literals so that `armlink` can merge them.

A literal pool is a block of memory embedded in the code to hold literal values. These values can be constants or long branch addresses.

`armclang` does not trade off literal pool sharing against unused section elimination. For example, you might have five functions in separate sections. You can keep the five functions in separate sections, so the linker can eliminate any that you did not use in your image. Therefore, the subset of the functions that are left in the link can still share their literals.

Also, `armclang` allows a global approach to literal-sharing. The linker can globally search for opportunities to share literals, even between functions from different parts of the code base that you might not have realised were using similar literals.

To make the best use of this feature, specify the `armclang` option `-ffunction-sections`, which is the default setting. The `-ffunction-sections` option does not affect the literal pool generation for a function. However, because the linker merging of literal pools only works on literal pools at the end of a section, `-ffunction-sections` gives the optimization more opportunities. The correct literal-merging behavior is visible only in the final image after linking, because the object files still contain the unmerged versions of the literals.

Options that affect literal pools

Although Arm® Compiler for Embedded FuSa 6 does not provide explicit literal pool generation options, the following are some examples of when literal pools get generated:

- `-Oz` can generate literal pools instead of the `MOVW` and `MOVT` pair of instructions, for improved code size. However, Cortex®-M0 does not support the `MOVW` and `MOVT` instructions, so it uses literal pools at all optimization levels.
- For processors that support M-profile architectures, such as Cortex-M3, you can use the `armclang` option `-mexecute-only`. Although this option disables literal pools and branch tables, the Arm libraries are built with literal pools. Therefore, libraries still use literal pools, even when you use the `-mexecute-only` option.

Related information

-ffunction-sections, -fno-function-sections
-mexecute-only
-O

5. Assembling Assembly Code

The Arm® Compiler for Embedded FuSa toolchain can assemble source code for both GNU syntax assembly language and `armasm` legacy assembly language.


Note

The `armasm` legacy assembler is deprecated, and it has not been updated since Arm Compiler 6.10. As a reminder, `armasm` always reports the deprecation warning `A1950W`. To suppress this message, specify the `--diag_suppress=1950` option.

`armasm` does not support:

- Armv8.4-A and later architectures.
- Armv8-R AArch64 targets.
- Certain backported options in Armv8.2-A and Armv8.3-A.
- Assembling Scalable Matrix Extension (SME) instructions.
- Assembling Scalable Vector Extension (SVE) instructions.
- Assembling Armv8.1-M or later architectures, M-profile Vector Extension (MVE).

5.1 Assembling GNU syntax and armasm assembly code

GNU and `armasm` are two different syntaxes for assembly language source code. They are similar, but have a number of differences. For example, GNU syntax identifies labels by the presence of a colon, while `armasm` syntax identifies them by their position at the start of a line.


Note

The *GNU Binutils - Using as* documentation provides complete information about GNU syntax assembly code.

The *Migration and Compatibility Guide* contains detailed information about the differences between GNU syntax and `armasm` syntax assembly to help you migrate legacy assembly code.

The following examples show equivalent GNU syntax and `armasm` assembly code for incrementing a register in a loop.

GNU assembler syntax

```
// Simple GNU syntax example
//
// Iterate round a loop 10 times, adding 1 to a register each time.

        .text
        .file "file.S"
        .section .text.main,"ax",@progbits
        .p2align     2
        .type    main,@function
```

```
main:
        MOV      w5,#0x64       // W5 = 100
        MOV      w4,#0          // W4 = 0
        B        test_loop      // branch to test_loop
loop:
        ADD      w5,w5,#1       // Add 1 to W5
        ADD      w4,w4,#1       // Add 1 to W4
test_loop:
        CMP      w4,#0xa        // if W4 < 10, branch back to loop
        BLT      loop
        .end
```

Use GNU syntax for newly created assembly files. Use the `armclang` integrated assembler to assemble GNU assembly language source code. Typically, you invoke the `armclang` assembler as follows:

```
armclang --target=aarch64-arm-none-eabi -c -o file.o file.S
```

armasm assembler syntax

```
; Simple armasm syntax example
;
; Iterate round a loop 10 times, adding 1 to a register each time.

        AREA ||.text||, CODE, READONLY, ALIGN=2

main PROC
        MOV      w5,#0x64       ; W5 = 100
        MOV      w4,#0          ; W4 = 0
        B        test_loop      ; branch to test_loop
loop
        ADD      w5,w5,#1       ; Add 1 to W5
        ADD      w4,w4,#1       ; Add 1 to W4
test_loop
        CMP      w4,#0xa        ; if W4 < 10, branch back to loop
        BLT      loop
        ENDP

        END
```

You might have legacy assembly source files that use the `armasm` syntax. Use `armasm` to assemble legacy `armasm` syntax assembly code. Typically, you invoke the `armasm` assembler as follows:

```
armasm --cpu=8-A.64 -o file.o file.s
```

Related information

GNU Binutils - Using as
Migrating armasm syntax assembly code to GNU syntax

5.2 How to get a backtrace through assembler functions

To backtrace through a function, a debugger must know how to calculate the return address. The `armclang` option `-g` inserts this information when generating assembly from C and C++ source code. For GNU-syntax assembly source code, you must add the information directly.

To debug Arm code, an Arm-compatible debugger expects the `.debug_frame` section to be present. Arm® Compiler for Embedded FuSa 6 exclusively uses `.debug_frame` to keep the code size small. There is a similarly formatted section called `.eh_frame`, used by the program itself for handling C++ exceptions. `armclang` does not include the `.eh_frame` section unless it is necessary.

The `armclang` integrated assembler does not automatically generate this information. Therefore, you must add the information into your GNU-syntax assembly code using `.cfi` directives.

Adding `.cfi` directives for functions that return using the link register (LR) is easy. Using directives to describe the location of variables in registers and the stack is more difficult. Because most assembler functions do not use the stack, only a backtrace is required. Therefore, you need only use a subset of the `.cfi` directives for most cases:

- `.cfi_sections .debug_frame`
- `.cfi_startproc`
- `.cfi_endproc`

To see where the `armclang` integrated assembler inserts the `.cfi` directives, compile the following C code:

```
// test.c
int main(void)
{
    return 0;
}
```

Compile `test.c` with:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a8 -g -O2 -S -o test.s test.c
```

`-g` generates the `.cfi` directives. `-O2` removes all use of the stack from `main()`. The `armclang` integrated assembler generates the following assembly:

```
    ...
main:
.Lfunc_begin0:
    .file   1 "<source_code_location>" "test.c"
    .loc    1 1 0
    .fnstart
    .cfi_sections .debug_frame
    .cfi_startproc
    .loc    1 1 18 prologue_end
    mov  r0, #0
    bx   lr
.Ltmp0:
.Lfunc_end0:
    .size   main, .Lfunc_end0-main
```

```
    .cfi_endproc
    .cantunwind
    .fnend
    ...
```

The function does not use the stack and returns using LR, so the `.cfi_startproc`, `.cfi_endproc`, and `.cfi_sections .debug_frame` directives are sufficient.

Functions that do not return using LR require more directives to tell the debugger that the return address is no longer in LR. For example:

```
    mov r1, lr // r1 = lr
    mov lr, #0 // use lr for something else.
    bx  r1 // return using r1
```

Here, more directives are needed after the `mov lr, #0` instruction. For the complete set of `.cfi` directives, see CFI directives.

Related information

Call Frame Information directives

5.3 Preprocessing assembly code

The C preprocessor must resolve assembly code that contains C preprocessor directives, for example `#include` or `#define`, before assembling.

By default, `armclang` uses the assembly code source file suffix to determine whether to run the C preprocessor:

- The `.s` (lowercase) suffix indicates assembly code that does not require preprocessing.
- The `.s` (uppercase) suffix indicates assembly code that requires preprocessing.

The `-x` option lets you override the default by specifying the language of the subsequent source files, rather than inferring the language from the file suffix. Specifically, `-x assembler-with-cpp` indicates that the assembly code contains C preprocessor directives and `armclang` must run the C preprocessor. The `-x` option only applies to input files that follow it on the command line.


Note

Do not confuse the `.ifdef` assembler directive with the preprocessor `#ifdef` directive:

- The preprocessor `#ifdef` directive checks for the presence of preprocessor macros. These macros are defined using the `#define` preprocessor directive or the `armclang` command-line option `-D`.
- The `armclang` integrated assembler `.ifdef` directive checks for code symbols. These symbols are defined using labels or the `.set` directive.

The preprocessor runs first and performs textual substitutions on the source code. This stage is when the `#ifdef` directive is processed. The source code is then passed onto the assembler, when the `.ifdef` directive is processed.

To preprocess an assembly code source file, do one of the following:

- Ensure that the assembly code filename has a `.s` suffix.

  For example:

  ```
  armclang --target=arm-arm-none-eabi -march=armv8-a test.S
  ```

- Use the `-x assembler-with-cpp` option to tell `armclang` that the assembly source file requires preprocessing. This option is useful when you have existing source files with the lowercase extension `.s`.

  For example:

  ```
  armclang --target=arm-arm-none-eabi -march=armv8-a -x assembler-with-cpp test.s
  ```


Note

If you want to preprocess assembly files that contain legacy `armasm`-syntax assembly code, then you must either:

- Use the `.s` filename suffix.
- Use separate steps for preprocessing and assembling.

For more information, see Command-line options for preprocessing assembly source code in the *Migration and Compatibility Guide*.

Related information

Command-line options for preprocessing assembly source code
-E (armclang)
-x (armclang)

6. Using Assembly and Intrinsics in C or C++ Code

All code for a single application can be written in the same source language. This source language is usually a high-level language such as C or C++ that is compiled to instructions for Arm® architectures. However, in some situations you might need lower-level control than that provided by C or C++.

For example:

- To access features that are not available from C or C++, such as interfacing directly with device hardware.
- To generate highly optimized code by using intrinsics or inline assembly to write sections of your code.

There are several ways to have low-level control over the generated code:

- Intrinsics are functions that the compiler provides. An intrinsic function has the appearance of a function call in C or C++, but compilation replaces the intrinsic by a specific sequence of low-level instructions.

  
  Note

  Arm compilers recognize Arm intrinsics, but are not guaranteed to work with any third-party compiler toolchains.

- Inline assembly lets you write assembly instructions directly in your C/C++ code, without the overhead of a function call.
- Calling assembly functions from C/C++ lets you write standalone assembly code in a separate source file. This code is assembled separately to the C/C++ code, and then integrated at link time.

6.1 Using intrinsics

Compiler intrinsics are special functions with implementations that are known to the compiler. These intrinsics enable you to easily incorporate domain-specific operations in C and C++ source code without resorting to complex implementations in assembly language.

The C and C++ languages are suited to many tasks but they do not provide built-in support for specific areas of application, for example Digital Signal Processing (DSP).

In a given application domain, there is usually a range of domain-specific operations that have to be performed frequently. However, if specific hardware support is available, then these operations can often be implemented more efficiently using the hardware support rather than in C or C++.

Using compiler intrinsics, you can achieve more complete coverage of target architecture instructions than you might get from the instruction selection of the compiler.

An intrinsic function has the appearance of a function call in C or C++, but compilation replaces the intrinsic by a specific sequence of low-level instructions.

Using compiler intrinsics offers some performance benefits:

- The low-level instructions substituted for an intrinsic are either as efficient as, or more efficient than, corresponding implementations in C or C++. The substitution results in both reduced instruction and cycle counts. To implement the intrinsic, the compiler automatically generates the best sequence of instructions for the specified target architecture. For example, the `__qadd` intrinsic maps directly to the A32 assembly language instruction `qadd`:

  ```
      QADD r0, r0, r1    ; Assuming r0 = a, r1 = b on entry
  ```

- More information is given to the compiler than the underlying C and C++ language is able to convey. This information enables the compiler to perform optimizations and to generate instruction sequences that it cannot otherwise perform.

These performance benefits can be significant for real-time processing applications. However, care is required because the use of intrinsics can decrease code portability.

Some intrinsics are necessary because the compiler does not otherwise recognize them. For many cases, C code without intrinsics might be more efficient, more portable, and easier for the compiler to optimize. When the compiler can create the instruction you require, C code without intrinsics might be the better alternative.

Example: C code that can be replaced with an intrinsic

A typical example is the saturating add of two 32-bit signed two's complement integers, commonly used in DSP programming. The following example shows one way of writing a C implementation:

```
#include <limits.h>

int L_add(const int a, const int b)
{
    int c;
    c = (unsigned int)a + b;
    if (((a ^ b) & INT_MIN) == 0)
    {
        if ((c ^ a) & INT_MIN)
        {
            c = (a < 0) ? INT_MIN : INT_MAX;
        }
    }
    return c;
}
```

1. Compile with, for example:

   ```
   armclang --target=arm-arm-none-eabi -mcpu=cortex-m55 -S L_add.c
   ```

   ```
   ...
   ```

```
L_add:
...
    adds    r2, r1, r0
    eor.w   r3, r2, r0
    eors    r1, r0
    cmp.w   r3, #-1
    mov     r3, r2
    mvn     r12, #-2147483648
    it      le
    eorle.w r3, r12, r0, asr #31
    cmp     r1, #0
    csel    r0, r2, r3, mi
    bx  lr
...
```

2. To use the `__qadd` intrinsic, modify this example as follows:

```
#include <arm_acle.h>    /* Include ACLE intrinsics */

int saturating_add(int a, int b)
{
    return __qadd(a, b);  /* Saturated add of a and b */
}
```

3. Compile with:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m55 -S saturating_add.c
```

   This command generates the following assembly:

```
...
saturating_add:
...
    qadd    r0, r0, r1
    bx  lr
...
```

Example: C code that the compiler can convert to the required instruction

The previous example of the C implementation for a saturating add operation can be rewritten so that the compiler can create the required `qadd` instruction directly:

```
// qadd.c
#include <limits.h>

int qadd(int a, int b)
{
    long long c = (long long)a + b;
    if (c < INT_MIN) c = INT_MIN;
    if (c > INT_MAX) c = INT_MAX;
    return c;
}
```

Compile with, for example:

```
armclang -O3 --target=arm-arm-none-eabi -mcpu=cortex-m55 -S qadd.c
```

This command generates the following assembly:

```
...
qadd:
   ...
    qadd    r0, r0, r1
    bx  lr
...
```

Related information

Compiler-specific intrinsics
ACLE support
NEON Programmer's Guide

6.2 Custom Datapath Extension support

Arm C Language Extensions (ACLE) intrinsics for Custom Datapath Extension (CDE) are defined in the `arm_cde.h` system header.

These intrinsics are documented in the *Custom Datapath Extension* section of the Arm C Language Extensions document.

Example

The following example shows how to use the ACLE intrinsics for CDE:

1. Create the `foo.c` file containing the following code:

```
#include <arm_cde.h>

uint32_t foo(uint32_t source_register)
{
    return __arm_cx2(0, source_register, 4);
}
```

   In this file, the function `foo()` uses the `__arm_cx2()` ACLE intrinsic for CDE. This intrinsic generates a `CX2` instruction.

   A `CX2` instruction is a Custom class 2 instruction that computes a value based on a source register, an immediate, optionally the original value of the destination register, and also writes the result to the destination register.

   For example, the instruction `CX2 p0, r0, r1, #2` sends the immediate 2 and the register R1 to the CDE coprocessor p0, and writes the result returned by p0 to the register R0.

   The intrinsic is defined as follows:

```
uint32_t __arm_cx2(int coproc, uint32_t n, uint32_t imm);
```

   Where:

- `coproc` is the CDE coprocessor number to use.
- `n` is the variable to send to the CDE coprocessor via the general-purpose source register operand.
- `imm` is the compile-time constant immediate value to use.

This intrinsic generates a variant of the `CX2` instruction that does not use the destination register value to compute the result.

2. Compile `foo.c` with the command:

```
armclang --target=arm-arm-none-eabi -march=armv8.1-m.main+cdecp0 -O1 -c foo.c -o
 foo.o
```

The compiler generates a `CX2` instruction with the expected operands, and returns the result of the instruction in register R0.

3. Run the following fromelf command to examine the output:

```
fromelf --cpu=8.1-M.Main --coproc0=cde --text -c foo.o
```

```
...

** Section #3 '.text.foo' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 6 bytes (alignment 4)
    Address: 0x00000000

    $t.0
    [Anonymous symbol #3]
    foo
        0x00000000:    ee400004    @...    CX2      p0,r0,r0,#4
        0x00000004:    4770        pG      BX       lr
...
```

Related information

-march
-mcpu
--coprocN=value (fromelf)
ARM v8-M Supplement - CDE Reference Manual

6.3 Writing inline assembly code

The compiler provides an inline assembler that enables you to write assembly code in your C or C++ source code, for example to access features of the target processor that are not available from C or C++.

The `__asm` keyword can incorporate inline assembly code into a function using the GNU inline assembly syntax. For example:

```
#include <stdio.h>
```

```
int add(int i, int j)
{
  int res = 0;
  __asm ("ADD %[result], %[input_i], %[input_j]"
    : [result] "=r" (res)
    : [input_i] "r" (i), [input_j] "r" (j)
  );
  return res;
}

int main(void)
{
  int a = 1;
  int b = 2;
  int c = 0;

  c = add(a,b);

  printf("Result of %d + %d = %d\n", a, b, c);
}
```

Note

The inline assembler does not support legacy assembly code written in `armasm` assembler syntax. See the Migration and Compatibility Guide for more information about migrating `armasm` syntax assembly code to GNU syntax.

Using inline assembly rather than writing a separate `.s` file has the following advantages:

- Shifts the burden of handling the procedure call standard (PCS) from the programmer to the compiler. This includes allocating the stack frame and preserving all necessary callee-saved registers.
- Inline assembly code gives the compiler more information about what the assembly code does.
- The compiler can inline the function that contains the assembly code into its callers.
- Inline assembly code can take immediate operands that depend on C-level constructs, such as the size of a structure or the byte offset of a particular structure field.

Structure of an inline assembly statement

The general form of an `__asm` inline assembly statement is:

```
__asm [volatile] (code); /* Basic inline assembly syntax */
```

```
/* Extended inline assembly syntax */
__asm [volatile] (code_template
    : outputs
   [: inputs
   [: clobber_list]]
);
```

Use the `volatile` qualifier for assembler instructions that have processor side-effects, which the compiler might be unaware of. The `volatile` qualifier disables certain compiler optimizations, which might otherwise lead to the compiler removing the code block. The `volatile` qualifier is optional, but consider using it around your assembly code blocks to ensure the compiler does not remove them when compiling with `-O1` or higher.

code

The assembly instruction, for example `"ADD R0, R1, R2"`.

code_template

A template for an assembly instruction, for example `"ADD %[result], %[input_i], %[input_j]"`.

If you specify a `code_template` rather than `<code>` then you must specify the `outputs` before specifying the optional `inputs` and `clobber_list`.

outputs

A list of output operands, separated by commas. Each operand consists of a symbolic name in square brackets, a constraint string, and a C expression in parentheses. In this example, there is a single output operand: `[result] "=r" (res)`. The list can be empty. For example:

```
__asm ("ADD R0, %[input_i], %[input_j]"
  :  /* This is an empty output operand list */
  : [input_i] "r" (i), [input_j] "r" (j)
);
```

inputs

An optional list of input operands, separated by commas. Input operands use the same syntax as output operands. In this example, there are two input operands: `[input_i] "r" (i), [input_j] "r" (j)`. The list can be empty.

clobber_list

A comma-separated list of strings. Each string is the name of a register that the assembly code potentially modifies, but for which the final value is not important. To prevent the compiler from using a register for a template string in an inline assembly string, add the register to the clobber list.

For example, if a register holds a temporary value, include it in the clobber list. The compiler avoids using a register in this list as an input or output operand, or using it to store another value when the assembly code is executed.

The list can be empty. In addition to registers, the list can also contain special arguments:

"cc"

The instruction modifies the condition code flags.

"memory"

The instruction accesses unknown memory addresses.

The registers in `clobber_list` must use lowercase letters rather than uppercase letters. An example instruction with a `clobber_list` is:

```
__asm ("ADD R0, %[input_i], %[input_j]"
  :  /* This is an empty output operand list */
  : [input_i] "r" (i), [input_j] "r" (j)
  : "r5","r6","cc","memory" /*Use "r5" instead of "R5" */
);
```

Defining symbols and labels

You can use inline assembly to define symbols. For example:

```
__asm (".global __use_no_semihosting\n\t");
```

To define labels, use `:` after the label name. For example:

```
__asm ("my_label:\n\t");
```

Multiple instructions

You can write multiple instructions within the same `__asm` statement. This example shows an interrupt handler written in one `__asm` statement for an Arm®v8-M mainline architecture.

```
void HardFault_Handler(void)
{
  __asm (
    "TST LR, #0x40\n\t"
    "BEQ from_nonsecure\n\t"
    "from_secure:\n\t"
    "TST LR, #0x04\n\t"
    "ITE EQ\n\t"
    "MRSEQ R0, MSP\n\t"
    "MRSNE R0, PSP\n\t"
    "B hard_fault_handler_c\n\t"
    "from_nonsecure:\n\t"
    "MRS R0, CONTROL_NS\n\t"
    "TST R0, #2\n\t"
    "ITE EQ\n\t"
    "MRSEQ R0, MSP_NS\n\t"
    "MRSNE R0, PSP_NS\n\t"
    "B hard_fault_handler_c\n\t"
  );
}
```

Copy the above handler code to `file.c` and then you can compile it using:

```
armclang --target=arm-arm-none-eabi -march=armv8-m.main -S file.c -o file.s
```

Embedded assembly

You can write embedded assembly using `__attribute__((naked))`. For more information, see the reference page in the *Arm Compiler for Embedded FuSa Reference Guide*.

Related information

armclang Inline Assembler
Migrating armasm syntax assembly code to GNU syntax
Semihosting for AArch32 and AArch64

6.4 Calling assembly functions from C and C++

Often, all the code for a single application is written in the same source language. This is usually a high-level language such as C or C++. That code is then compiled to Arm assembly code.

However, in some situations you might want to make function calls from C/C++ code to assembly code. For example:

- If you want to make use of existing assembly code, but the rest of your project is in C or C++.
- If you want to manually write critical functions directly in assembly code that can produce better optimized code than compiling C or C++ code.
- If you want to interface directly with device hardware and if this is easier in low-level assembly code than high-level C or C++.


Note

For code portability, it is better to use intrinsics or inline assembly rather than writing and calling assembly functions.

To call an assembly function from C or C++:

1. In the assembly source, declare the code as a global function using `.global` and `.type`:

```
     .global  myadd
     .p2align 2
     .type    myadd,%function

myadd:                      // Function "myadd" entry point.
     .fnstart
     add      r0, r0, r1    // Function arguments are in R0 and R1. Add together
                            // and put the result in R0.
     bx       lr            // Return by branching to the address in the link
                            // register.
     .fnend
```


Note

`armclang` requires that you explicitly specify the types of exported symbols using the `.type` directive. If the `.type` directive is not specified in the above example, the linker outputs warnings of the form:

```
Warning: L6437W: Relocation #RELA:1 in test.o(.text) with respect to
myadd...

Warning: L6318W: test.o(.text) contains branch to a non-code symbol
myadd.
```

2. In C code, declare the external function using `extern`:

```
#include <stdio.h>

extern int myadd(int a, int b);
```

```
int main()
{
  int a = 4;
  int b = 5;
  printf("Adding %d and %d results in %d\n", a, b, myadd(a, b));
  return (0);
}
```

In C++ code, use `extern "C"`:

```
extern "C" int myadd(int a, int b);
```

3. Ensure that your assembly code complies with the *Procedure Call Standard for the Arm Architecture* (AAPCS).

   The AAPCS describes a contract between caller functions and callee functions. For example, for integer or pointer types, it specifies that:

   - Registers R0-R3 pass argument values to the callee function, with subsequent arguments passed on the stack.
   - Register R0 passes the result value back to the caller function.
   - Caller functions must preserve R0-R3 and R12, because these registers are allowed to be corrupted by the callee function.
   - Callee functions must preserve R4-R11 and LR, because these registers are not allowed to be corrupted by the callee function.

   For more information, see the Application Binary Interface (ABI) documentation.
4. Compile both source files:

```
armclang --target=arm-arm-none-eabi -march=armv8-a main.c myadd.s
```

Related information

Procedure Call Standard for the Arm Architecture
Procedure Call Standard for the Arm 64-bit Architecture

7. SVE Coding Considerations with Arm Compiler for Embedded FuSa 6

The Arm® Compiler for Embedded FuSa toolchain supports targets that implement the Scalable Vector Extension (SVE) for Armv8-A AArch64.

SVE is a SIMD instruction set for AArch64, that introduces the following architectural features for High Performance Computing (HPC):

- Scalable vector length.
- Per-lane predication.
- Gather-load and scatter-store.
- Fault-tolerant speculative vectorization.
- Horizontal and serialized vector operations.

This release of the Arm Compiler for Embedded FuSa toolchain lets you:

- Assemble source code containing SVE instructions.
- Disassemble ELF object files containing SVE instructions.
- Compile C and C++ code for SVE-enabled targets.
- Use intrinsics to write SVE instructions directly from C code.


Note

The Arm Compiler for Embedded FuSa toolchain only supports bare-metal applications. For SVE compilation for Linux, use Arm Compiler for Linux. For more information, see Arm Compiler for Linux.


Note

Arm Compiler for Embedded FuSa supports auto-vectorization for SVE, but does not include SVE-optimized libraries. Suitable SVE-optimized libraries are supplied with Arm Compiler for Linux. For more information, see Arm Compiler for Linux.

7.1 Assembling SVE code

Use `armclang` with a suitable SVE-enabled target to assemble code containing SVE instructions.

The SVE architectural extension to the Arm®v8-A architecture (`armv8-a+sve`) provides SVE instructions. Many of these SVE instructions make use of the `p` and `z` register classes.

The following example shows a simple assembly program that includes SVE instructions.

```
// example1.s
    .global main
main:
    mov     x0, 0x90000000
    mov     x8, xzr
    ptrue   p0.s                        //SVE instruction
    fcpy    z0.s, p0/m, #5.00000000     //SVE instruction
    orr     w10, wzr, #0x400
loop:
    st1w    z0.s, p0, [x0, x8, lsl #2]  //SVE instruction
    incw    x8                          //SVE instruction
    whilelt p0.s, x8, x10               //SVE instruction
    b.any   loop                        //SVE instruction
    mov     w0, wzr
    ret
```

To assemble this source file into a binary object file, use `armclang` with an SVE-enabled target:

```
armclang -c --target=aarch64-arm-none-eabi -march=armv8-a+sve example1.s -o example1.o
```

The command-line options in this example are:

`-c`

Instructs the compiler to perform the compilation step, but not the link step.

`--target=aarch64-arm-none-eabi`

Instructs the compiler to generate A64 instructions for AArch64 state.


Note

SVE is not supported with AArch32 state, so the `--target=aarch64-arm-none-eabi` option is mandatory.

`-march=armv8-a+sve`

Specifies that the compiler targets the Armv8-A architecture profile with the SVE target feature enabled.

The default for AArch64 is `-march=armv8-a`, that is the Armv8-A architecture profile without the SVE extension. You must explicitly specify `+sve` to assemble SVE instructions.

Armv8-A and later architectures support the SVE extension. For example, `-march=armv8.1-a+sve`.

`example1.s`

Input assembly language file.

`-o example1.o`

Output ELF object file.

Related information

Disassembling SVE object files on page 131

Arm Compiler for Embedded FuSa Reference Guide
-c (armclang)
-o (armclang)
-march (armclang)
--target (armclang)

7.2 Disassembling SVE object files

Use the `fromelf` tool without specifying `--cpu` to display the details and contents of an ELF-format binary file. This includes disassembly of the code sections of an object containing SVE instructions.

About this task

To disassemble an ELF-format object file containing SVE instructions, use `fromelf` with the `-c` option.

Procedure

1. Use the C file `matmul_f64_sve.c` from the example in Running a binary in an AEMv8-A Base Fixed Virtual Platform (FVP).
2. Compile and use `fromelf` to view the disassembly:

```
armclang -c -O3 --target=aarch64-arm-none-eabi -march=armv8-a+sve -o
 matmul_f64_sve.o matmul_f64_sve.c
fromelf -c matmul_f64_sve.o
```

The disassembly is as follows:

```
...
** Section #3 '.text.matmul_f64_sve' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 432 bytes (alignment 4)
    Address: 0x00000000

    $x.0
    matmul_f64_sve
        0x00000000:    fc1a0fea    ....    STR      d10,[sp,#-0x60]!
        0x00000004:    a90457f6    .W..    STP      x22,x21,[sp,#0x40]
        0x00000008:    aa0003f5    ....    MOV      x21,x0
        0x0000000c:    04e0e3f6    ....    CNTD     x22
        0x00000010:    90000000    ....    ADRP     x0,{pc} ; 0x10
        ...
        0x00000190:    54fffe43    C..T    B.CC     {pc}-0x38 ; 0x158
        0x00000194:    a9454ff4    .OE.    LDP      x20,x19,[sp,#0x50]
        0x00000198:    a94457f6    .WD.    LDP      x22,x21,[sp,#0x40]
        0x0000019c:    a9435ff8    ._C.    LDP      x24,x23,[sp,#0x30]
        0x000001a0:    a94267fe    .gB.    LDP      x30,x25,[sp,#0x20]
        0x000001a4:    6d4123e9    .#Am    LDP      d9,d8,[sp,#0x10]
        0x000001a8:    fc4607ea    ..F.    LDR      d10,[sp],#0x60
        0x000001ac:    d65f03c0    .._.    RET
...
```

Related information

Assembling SVE code on page 129

7.3 Running a binary in an AEMv8-A Base Fixed Virtual Platform (FVP)

Describes how to compile a program with Arm® Compiler for Embedded FuSa and then run the resulting binary using the AEMvA Base Fixed Virtual Platform (FVP). The examples use various SVE intrinsics.

Running the FVP

The command to execute a compiled binary through the FVP is fairly complex, but there are only a few elements that can be edited.

The following example shows a complete command-line invocation of the FVP. Most of the lines are required for correct program execution and do not need to be modified. `$VECLEN`, `$CMDLINE`, and `$BINARY` are parameters that can be edited.

```
$FVP_BASE/FVP_Base_AEMvA \

  --plugin $FVP_BASE/ScalableVectorExtension.so \
  -C SVE.ScalableVectorExtension.veclen=$VECLEN \
  --quiet \
  --stat \
  -C cluster0.NUM_CORES=1 \
  -C bp.secure_memory=0 \
  -C bp.refcounter.non_arch_start_at_default=1 \
  -C cluster0.cpu0.semihosting-use_stderr=1 \
  -C bp.vis.disable_visualisation=1 \
  -C cluster0.cpu0.semihosting-cmd_line="$CMDLINE" \
  -a cluster0.cpu0=$BINARY
```

Where:

`$FVP_BASE`
: Specifies the path to the FVP.

`$VECLEN`
: Defines the SVE vector width, in units of 64-bit (8 byte) blocks. The maximum value is 32, which corresponds to the architectural maximum SVE vector width of 2048 bits (256 bytes).

  The SVE architecture only supports vector lengths in 128-bit (16 byte increments), so all values of `$VECLEN` must be even. For example, a value of 8 signifies a 512-bit vector width.

`--quiet`
: Specifies that the FVP emits reduced output. For example, if `--quiet` is omitted, `Simulation is started` and `Simulation is terminating` messages are output to signify the start and end of program execution.

`--stat`
: Specifies that the FVP writes a short summary of program execution to standard output following termination (even if `--quiet` is specified).

This output is of the form:

```
--- FVP_Base_AEMvA statistics: -----------------------------------------------
Simulated time                          : 0.039700s
User time                               : 2.234375s
System time                             : 0.000000s
Wall time                               : 2.233020s
Performance index                       : 0.02
FVP_Base_AEMvA.cluster0.cpu0            :   1.78 MIPS (     3980000 Inst)
------------------------------------------------------------------------------
```

`$CMDLINE`

Specifies the command line to pass to your program. This command line is typically of the form `"./<binary_name> <arg1> <arg2>"`.

`$BINARY`

Specifies the path to the compiled binary that the FVP is to load and execute.

A sample application

The following sample application, `matmul_f64_sve.c`, is derived from the `matmul_f64` example provided in SVE Programming Examples, and uses the `svcntd`, `svdup_f64`, `svld1`, `svld1rq`, and `svmla_lane` SVE intrinsics:

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <math.h>
#include <time.h>
#include <arm_sve.h>

typedef double float64_t;

#define A 128
#define B 128
#define C 128

void matmul_f64_sve( uint64_t M, uint64_t K, uint64_t N,
    float64_t * inLeft, float64_t * inRight, float64_t * out) {
    uint64_t x, y, z;
    svbool_t p64_all = svptrue_b64();
    uint64_t vl = svcntd();
    uint64_t offsetIN_1, offsetIN_2, offsetIN_3;
    uint64_t offsetOUT_1, offsetOUT_2, offsetOUT_3;

    float64_t *ptrIN_left;
    float64_t *ptrIN_right;
    float64_t *ptrOUT;

    svfloat64_t acc0, acc1, acc2, acc3;
    svfloat64_t inR_0, inR_1;
    svfloat64_t inL_0, inL_1, inL_2, inL_3;

    offsetIN_1 = K;
    offsetIN_2 = 2*K;
    offsetIN_3 = 3*K;

    offsetOUT_1 = N;
    offsetOUT_2 = 2*N;
    offsetOUT_3 = 3*N;

    for (x=0; x<M; x+=4) {
        ptrOUT = &out[x*N];
```

```
        for (y=0; y<N; y+=vl) {
            acc0 = svdup_f64(0.0);
            acc1 = svdup_f64(0.0);
            acc2 = svdup_f64(0.0);
            acc3 = svdup_f64(0.0);

            ptrIN_left = &inLeft[x*K];
            ptrIN_right = &inRight[y];

            for (z=0; z<K; z+=2) {
                inR_0 = svld1(p64_all, ptrIN_right);
                inR_1 = svld1(p64_all, &ptrIN_right[offsetOUT_1]);

                inL_0 = svld1rq(p64_all, ptrIN_left);
                inL_1 = svld1rq(p64_all, &ptrIN_left[offsetIN_1]);
                inL_2 = svld1rq(p64_all, &ptrIN_left[offsetIN_2]);
                inL_3 = svld1rq(p64_all, &ptrIN_left[offsetIN_3]);

                acc0 = svmla_lane(acc0, inR_0, inL_0, 0);
                acc0 = svmla_lane(acc0, inR_1, inL_0, 1);

                acc1 = svmla_lane(acc1, inR_0, inL_1, 0);
                acc1 = svmla_lane(acc1, inR_1, inL_1, 1);

                acc2 = svmla_lane(acc2, inR_0, inL_2, 0);
                acc2 = svmla_lane(acc2, inR_1, inL_2, 1);

                acc3 = svmla_lane(acc3, inR_0, inL_3, 0);
                acc3 = svmla_lane(acc3, inR_1, inL_3, 1);

                ptrIN_right += 2*N;
                ptrIN_left += 2;
            }

            svst1(p64_all, ptrOUT, acc0);
            svst1(p64_all, &ptrOUT[offsetOUT_1], acc1);
            svst1(p64_all, &ptrOUT[offsetOUT_2], acc2);
            svst1(p64_all, &ptrOUT[offsetOUT_3], acc3);

            ptrOUT += vl;
        }
    }
}

// Disable all SVE traps by setting CPTR_EL3.EZ bit [8] and clearing CPTR_EL3.TFP
 bit [10]
void disable_sve_traps(void)
{
    __asm(
    "MRS x0, CPTR_EL3\n"
    "BIC x0, x0, #(1<<10)\n"
    "ORR x0, x0, #(1<<8)\n"
    "MSR CPTR_EL3, x0\n"
    "ISB\n"
    );
}

int main(int argc, char* argv[]) {
    float64_t inLeft[A*B];
    float64_t inRight[B*C];

    float64_t out[A*C] = {0};

    printf("\nSVE Matrix Multiply Float64 example\n");

    disable_sve_traps();

    srand((unsigned int)time(0));

    for(int64_t x = 0; x < (A * B); ++x)
```

```
    {
        inLeft[x] = ((double)(rand() % 2000000) / 100.f) - 10000.0;
    }
    for(int64_t x = 0; x < (B * C); ++x)
    {
        inRight[x] = ((double)(rand() % 2000000) / 100.f) - 10000.0;
    }

    matmul_f64_sve(A, B, C, inLeft, inRight, out);
    return 0;
}
```

Note

The `arm_sve.h` header file is not supported for big-endian targets.

Note

The `disable_sve_traps()` function is required on hardware to configure the EZ and TFP bits in CPTR_EL3 by default to trap execution of SVE or SVE2 instructions. For more details, see CPTR_EL3, Architectural Feature Trap Register (EL3).

For FVP models, you can either use the `disable_sve_traps()` function or specify the `-C SVE.ScalableVectorExtension.enable_at_reset=true` parameter.

To compile this application and create an executable binary:

```
armclang -O3 -Xlinker "--ro_base=0x80000000" --target=aarch64-arm-none-eabi -
march=armv8-a+sve -o matmul_f64_sve.axf matmul_f64_sve.c
```

Running the sample application on an FVP

To execute an application using an FVP, it is useful to construct a shell script as follows:

```
#!/bin/bash

# fvp-run.sh
# Usage: fvp-run.sh [veclen] [binary]
#    Executes the specified binary in the FVP, with no command-line
#    arguments. The SVE register width is [veclen] x 64 bits. Only
#    even values of veclen are valid.
#
#
# Set the FVP_BASE environment variable to point to the FVP directory.

VECLEN=$1
CMDLINE=$2

$FVP_BASE/FVP_Base_AEMvA \
   --plugin $FVP_BASE/ScalableVectorExtension.so \
   -C SVE.ScalableVectorExtension.veclen=$VECLEN \
   --quiet \
   --stat \
   -C cluster0.NUM_CORES=1 \
   -C bp.secure_memory=0 \
   -C bp.refcounter.non_arch_start_at_default=1 \
   -C cluster0.cpu0.semihosting-use_stderr=1 \
   -C bp.vis.disable_visualisation=1 \
   -C cluster0.cpu0.semihosting-cmd_line="$CMDLINE"  \
```

```
    -a cluster0.cpu0=$CMDLINE
```

This script loads and executes the compiled binary with the FVP, and outputs the following information:

```
terminal_0: Listening for serial connection on port 5000
terminal_1: Listening for serial connection on port 5001
terminal_2: Listening for serial connection on port 5002
terminal_3: Listening for serial connection on port 5003

SVE Matrix Multiply Float64 example

Info: /OSCI/SystemC: Simulation stopped by user.

--- FVP_Base_AEMvA statistics: -------------------------------------------------
Simulated time                          : 0.040400s
User time                               : 0.312500s
System time                             : 0.000000s
Wall time                               : 0.315253s
Performance index                       : 0.13
FVP_Base_AEMvA.cluster0.cpu0            :  12.93 MIPS (     4040115 Inst)
--------------------------------------------------------------------------------
```

The statistics values might be different from those shown here.

Related information

Arm Compiler for Embedded FuSa Reference Guide
-o (armclang)
armclang -Xlinker option
armclang -Olevel option
-march (armclang)
--target (armclang)

7.4 Embedding SVE assembly code directly into C and C++ code

The inline assembly mechanism lets you vectorize parts of a function by hand without having to write the entire function in assembly code.


Note

This information assumes that you are familiar with details of the SVE Architecture, including vector-width agnostic registers, predication, and `WHILE` operations.

The following sections describe information relating to SVE. For general information about writing inline assembly code, see Writing inline assembly code.

Outputs

Each entry in outputs has one of the following forms:

```
[name] "=&register-class" (destination)
[name] "=register-class" (destination)
```

The first form has the register class preceded by `=&`. This form specifies that the assembly instructions might read from one of the inputs (specified in the `inputs` section of the `__asm` statement) after writing to the output.

The second form has the register class preceded by `=`. This form specifies that the assembly instructions never read from inputs in this way. Using the second form is an optimization. It allows the compiler to allocate the same register to the output as it allocates to one of the inputs.

Both forms specify that the assembly instructions produce an output that is stored in the C object specified by `destination`. This can be any scalar value that is valid for the left-hand side of a C assignment. The register-class field specifies the type of register that the assembly instructions require. It can be one of:

`r`

The register for this output when used within the assembly instructions is a general-purpose register (x0-x30)

`w`

The register for this output when used within the assembly instructions is a SIMD and floating-point register (v0-v31).

It is not possible at present for outputs to contain an SVE vector or predicate value. All uses of SVE registers must be internal to the inline assembly block.

It is the responsibility of the compiler to allocate a suitable output register and to copy that register into the `destination` after the `__asm` statement is executed. The assembly instructions within the instructions section of the `__asm` statement can use one of the following forms to refer to the output value:

`%[name]`

Refers to an r-class output as `x<N>` or a w-class output as `v<N>`.

`%w[name]`

Refers to an r-class output as `w<N>`.

`%s[name]`

Refers to a w-class output as `s<N>`.

`%d[name]`

Refers to a w-class output as `d<N>`.

In all cases `<N>` represents the number of the register that the compiler has allocated to the output. The use of these forms means that it is not necessary for the programmer to anticipate precisely which register is selected by the compiler. The following example creates a function that returns

the value 10. It shows how the programmer is able to use the `%w[res]` form to describe the movement of a constant into the output register without knowing which register is used.

```
int f()
{
  int result;
  __asm("movz %w[res], #10" : [res] "=r" (result));
  return result;
}
```

In optimized output the compiler picks the return register (0) for `res`, resulting in the following assembly code:

```
movz w0, #10
ret
```

Inputs

Within an `asm` statement, each entry in the `inputs` section has the form:

```
[<name>] "<operand-type>" (<value>)
```

This construct specifies that the `__asm` statement uses the scalar C expression value as an input, referred to within the assembly instructions as `name`. The `<operand-type>` field specifies how the input value is handled within the assembly instructions. It can be one of the following:

r
: The input is to be placed in a general-purpose register (x0-x30).

w
: The input is to be placed in a SIMD and floating-point register (v0-v31).

[<output-name>]
: The input is to be placed in the same register as output `<operand-type>`. In this case the `[<name>]` part of the input specification is redundant and can be omitted. The assembly instructions can use the forms described in Outputs to refer to both the input and the output. That is, `%[<name>]`, `%w[<name>]`, `%s[<name>]`, and `%d[<name>]`.

i
: The input is an integer constant and is used as an immediate operand. The assembly instructions use `%[<name>]` in place of immediate operand `<#N>`, where `<N>` is the numerical value of `<value>`.

In the first two cases, it is the responsibility of the compiler to allocate a suitable register and to ensure that it contains `<value>` on entry to the assembly instructions. The assembly instructions must refer to these registers using the same syntax as for the outputs. That is, `%[<name>]`, `%w[<name>]`, `%s[<name>]`, and `%d[<name>]`.

It is not possible at present for inputs to contain an SVE vector or predicate value. All uses of SVE registers must be internal to instructions.

This example shows an `__asm` directive with the same effect as the previous example, except that an i-form input is used to specify the constant to be assigned to the result.

```
int f()
{
   int result;
   __asm("movz %w[res], %[value]" : [res] "=r" (result) : [value] "i" (10));
   return result;
}
```

Side effects

Many `asm` statements have effects other than reading from inputs and writing to outputs. This is particularly true of `__asm` statements that implement vectorized loops, since most such loops read from or write to memory. The `<clobber_list>` section of an `__asm` statement tells the compiler what these additional effects are. Each entry must be one of the following:

`"memory"`
: The `__asm` statement reads from or writes to memory. This is necessary even if inputs contain pointers to the affected memory.

`"cc"`
: The `__asm` statement modifies the condition-code flags.

`"x<N>"`
: The `__asm` statement modifies general-purpose register `<N>`.

`"v<N>"`
: The `__asm` statement modifies SIMD and floating-point register `<N>`.

`"z<N>"`
: The `__asm` statement modifies SVE vector register `<N>`. Since SVE vector registers extend the SIMD and floating-point registers, this is equivalent to writing `"v<N>"`.

`"p<N>"`
: The `__asm` statement modifies SVE predicate register `<N>`.

Use of volatile

Sometimes an `__asm` statement might have dependencies and side effects that cannot be captured by the `__asm` statement syntax. For example, suppose there are three separate `__asm` statements (not three lines within a single `__asm` statement), that do the following:

- The first sets the floating-point rounding mode.
- The second executes on the assumption that the rounding mode set by the first statement is in effect.
- The third statement restores the original floating-point rounding mode.

It is important that these statements are executed in order, but the `__asm` statement syntax provides no direct method for representing the dependency between them. Instead, each statement must add the keyword `volatile` after `__asm`. This prevents the compiler from removing the `__asm` statement as dead code, even if the `__asm` statement does not modify memory and if

its results appear to be unused. The compiler always executes `__asm` volatile statements in their original order.

For example:

```
__asm volatile ("msr fpcr, %[flags]" :: [flags] "r" (new_fpcr_value));
```


Note

An `__asm volatile` statement must still have a valid side effects list. For example, an `__asm volatile` statement that modifies memory must still include `"memory"` in the side-effects section.

Labels

The compiler might output a given `__asm` statement more than once, either as a result of optimizing the function that contains the `__asm` statement or as a result of inlining that function into some of its callers. Therefore, `__asm` statements must not define named labels like `.loop`, since if the `__asm` statement is written more than once, the output contains more than one definition of label `.loop`. Instead, the assembler provides a concept of relative labels. Each relative label is simply a number and is defined in the same way as a normal label. For example, relative label 1 is defined by:

```
1:
```

The assembly code can contain many definitions of the same relative label. Code that refers to a relative label must add the letter `f` (forward) to refer the next definition or the letter `b` (backward) to refer to the previous definition. A typical assembly loop with a pre-loop test would therefore have the following structure:

```
    ...pre-loop test...
    b.none      2f
1:
    ...loop...
    b.any       1b
2:
```

This structure allows the compiler output to contain many copies of this code without creating any ambiguity.

Examples

The following example shows a simple function that performs a fused multiply-add operation (x=a·b +c) across four passed-in arrays of a size specified by `<n>`:

```
void f(double *restrict x, double *restrict a, double *restrict b,
       double *restrict c, unsigned long n)
{
  for (unsigned long i = 0; i < n; ++i)
  {
    x[i] = fma(a[i], b[i], c[i]);
  }
}
```

An `__asm` statement that exploits SVE instructions to achieve equivalent behavior might look like the following:

```
void f(double *x, double *a, double *b, double *c, unsigned long n)
{
  unsigned long i;
  __asm ("whilelo p0.d, %[i], %[n]             \n\
  1:                                           \n\
        ld1d z0.d, p0/z, [%[a], %[i], lsl #3]  \n\
        ld1d z1.d, p0/z, [%[b], %[i], lsl #3]  \n\
        ld1d z2.d, p0/z, [%[c], %[i], lsl #3]  \n\
        fmla z2.d, p0/m, z0.d, z1.d            \n\
        st1d z2.d, p0, [%[x], %[i], lsl #3]    \n\
        uqincd %[i]                            \n\
        whilelo p0.d, %[i], %[n]               \n\
        b.any 1b"
   : [i] "=&r" (i)
   : "[i]" (0),
     [x] "r" (x),
     [a] "r" (a),
     [b] "r" (b),
     [c] "r" (c),
     [n] "r" (n)
   : "memory", "cc", "p0", "z0", "z1", "z2");
}
```


Note

Keeping the `restrict` qualifiers would be valid but would have no effect.

The input specifier `"[i]" (0)` indicates that the assembly statements take an input 0 in the same register as output `[i]`. In other words, the initial value of `[i]` must be zero. The use of `=&` in the specification of `[i]` indicates that `[i]` cannot be allocated to the same register as `[x]`, `[a]`, `[b]`, `[c]`, or `[n]` (because the assembly instructions use those inputs after writing to `[i]`).

In this example, the C variable `i` is not used after the `__asm` statement. In effect the `__asm` statement is simply reserving a register that it can use as scratch space. Including `"memory"` in the side effects list indicates that the `__asm` statement reads from and writes to memory. The compiler must therefore keep the `__asm` statement even though `i` is not used.

7.5 Using SVE and SVE2 intrinsics directly in your C code

Intrinsics are C or C++ pseudo-function calls that the compiler replaces with the appropriate SIMD instructions. These intrinsics let you use the data types and operations available in the SIMD implementation, while allowing the compiler to handle instruction scheduling and register allocation.

These intrinsics are defined in the Arm C Language Extensions specification.

Introduction

The Arm C Language Extensions (ACLE) for SVE provide a set of types and accessors for SVE vectors and predicates, and a function interface for all relevant SVE and SVE2 instructions.

The function interface is more general than the underlying architecture, so not every function maps directly to an architectural instruction. The intention is to provide a regular interface and leave the compiler to pick the best mapping to SVE or SVE2 instructions.

The Arm C Language Extensions specification has a detailed description of this interface, and must be used as the primary reference. This section introduces a selection of features to help you get started with the ACLE for SVE.

Header file inclusion

Translation units that use the ACLE must first include `arm_sve.h`, guarded by `__ARM_FEATURE_SVE`:

```
#ifdef __ARM_FEATURE_SVE
#include <arm_sve.h>
#endif /* __ARM_FEATURE_SVE */
```

All functions and types that are defined in the header file have the prefix `sv`, to reduce the chance of collisions with other extensions.

Note

The `arm_sve.h` header file is not supported for big-endian targets.

SVE vector types

`arm_sve.h` defines the following C types to represent values in SVE vector registers. Each type describes the type of the elements within the vector:

`svint8_t svuint8_t`

`svint16_t svuint16_t svfloat16_t`

`svint32_t svuint32_t svfloat32_t`

`svint64_t svuint64_t svfloat64_t`

For example, `svint64_t` represents a vector of 64-bit signed integers, and `svfloat16_t` represents a vector of half-precision floating-point numbers.

SVE predicate type

The extension also defines a single sizeless predicate type `svbool_t`, which has enough bits to control an operation on a vector of bytes.

The main use of predicates is to select elements in a vector. When the elements in the vector have N bytes, only the low bit in each sequence of N predicate bits is significant, as shown in the following table:

Table 7-1: Element selection by predicate type svbool_t

Vector type	Element selected by each svbool_t bit									
`svint8_t`	0	1	2	3	4	5	6	7	8	...
`svint16_t`	0		1		2		3		4	...
`svint32_t`	0				1				2	...
`svint64_t`	0								1	...

Limitations on how SVE ACLE types can be used

SVE is a vector-length agnostic architecture, allowing an implementation to choose a vector length of any multiple of 128 bits, up to a maximum of 2048 bits. Therefore, the size of SVE ACLE types is unknown at compile time, which limits how these types can be used.

Common situations where SVE types might be used include:

- As the type of an object with automatic storage duration.
- As a function parameter or return type.
- As the type in a `(type) <value>` compound literal.
- As the target of a pointer or reference type.
- As a template type argument.

Because of their unknown size at compile time, SVE types must not be used:

- To declare or define a `static` or thread-local storage variable.
- As the type of an array element.
- As the operand to a `new` expression.
- As the type of object that is deleted by a `delete` expression.
- As the argument to `sizeof` and `_Alignof`.
- With pointer arithmetic on pointers to SVE objects (this affects the `+`, `-`, `++`, and `--` operators).
- As members of unions, structures and classes.
- In standard library containers like `std::vector`.

For a comprehensive list of valid usage, refer to the Arm C Language Extensions specification.

Calling SVE ACLE functions

SVE ACLE functions have the form:

```
sv<base>[_<disambiguator>][_<type0>][_<type1>]...[_<predication>]
```

Where the function is built using the following:

<base>

For most functions, this name is the lowercase name of the SVE instruction. Sometimes, letters indicating the type or size of data being operated on are omitted, where it can be implied from the argument types.

Unsigned extending loads add a `u` to indicate that the data is zero extended, to more explicitly differentiate them from their signed equivalent.

<disambiguator>

This field distinguishes between different forms of a function, for example:

- To distinguish between addressing modes
- To distinguish forms that take a scalar rather than a vector as the final argument.

<type0> <type1> ...

A list of types for vectors and predicates, starting with the return type then with each argument type. For example, `_s8`, `_u32`, and `_f32`, which represent signed 8-bit integer, an unsigned 32-bit integer and single-precision 32-bit float types, respectively.

Predicate types are represented by, for example, `_b8` and `_b16`, for predicates suitable for 8-bit and 16-bit types respectively. A predicate type suitable for all element types is represented by `_b`. Where a type is not needed to disambiguate between variants of a base function, it is omitted.

<predication>

This suffix describes the inactive elements in the result of a predicated operation. It can be one of the following:

- `z` - Zero predication: Set all inactive elements of the result to zero.
- `m` - Merge predication: copy all inactive elements from the first vector argument.
- `x` - 'Don't care' predication. Use this form when you do not care about the inactive elements. The compiler is then free to choose between zeroing, merging, or unpredicated forms to give the best code quality, but gives no guarantee of what data is left in inactive elements.

Addressing modes

Load, store, prefetch, and `ADR` functions have arguments that describe the memory area being addressed. The first addressing argument is the base - either a single pointer to an element type, or a 32-bit or 64-bit vector of addresses. The second argument, when present, offsets the base (or bases) by some number of bytes, elements, or vectors. This offset argument can be an immediate constant value, a scalar argument, or a vector of offsets.

Not every combination of the addressing modes exists. The following table gives examples of some common addressing mode disambiguators, and describes how to interpret the address arguments:

Table 7-2: Common addressing mode disambiguators

Disambiguator	Interpretation
`_u32base`	The base argument is a vector of unsigned 32-bit addresses.
`_u64base`	The base argument is a vector of unsigned 64-bit addresses.

Disambiguator	Interpretation

`_s32offset`
`_s64offset`
`_u32offset`
`_u64offset`
	The offset argument is a vector of byte offsets. These offsets are signed or unsigned 32-bit or 64-bit numbers.

`_s32index`
`_s64index`
`_u32index`
`_u64index`
	The offset argument is a vector of element-sized indices. These indices are signed or unsigned 32-bit or 64-bit numbers.

`_offset`	The offset argument is a scalar, and must be treated as a byte offset.
`_index`	The offset argument is a scalar, and must be treated as an index into an array of elements.
`_vnum`	The offset argument is a scalar, and must be treated an index into an array of SVE vectors.

In the following example, the address of element `i` is `&base[indices[i]]`.

```
svuint32_t svld1_gather_[s32]index[_u32]
    (svbool_t pg, const uint32_t *base, svint32_t indices)
```

Operations involving vectors and scalars

All arithmetic functions that take two vector inputs have an alternative form that takes a vector and a scalar. Conceptually, this scalar is duplicated across a vector, and that vector is used as the second vector argument.

Similarly, arithmetic functions that take three vector inputs have an alternative form that takes two vectors and one scalar.

To differentiate these forms, the disambiguator `_n` is added to the form that takes a scalar.

Short forms

Sometimes, it is possible to omit part of the full name, and still uniquely identify the correct form of a function, by inspecting the argument types. Where omitting part of the full name is possible, these simplified forms are provided as aliases to their fully named equivalents, and are used for preference in the rest of this document.

In the Arm C Language Extensions specification, the portion that can be removed is enclosed in square brackets. For example `svclz[_s16]_m` has the full name `svclz_s16_m`, and an overloaded alias, `svclz_m`.

SVE2 intrinsics

SVE2 builds on SVE to add data-processing instructions that bring the benefits of scalable long vectors to a wider class of applications. To enable only the base SVE2 instructions, use the `+sve2`

option with the `armclang` options `-march` or `-mcpu`. To enable additional optional SVE2 instructions, use the following armclang options:

- `+sve2-aes` to enable scalable vector forms of `AESD`, `AESE`, `AESIMC`, `AESMC`, `PMULLB`, and `PMULLT` instructions.
- `+sve2-bitperm` to enable the `BDEP`, `BEXT`, and `BGRP` instructions.
- `+sve2-sha3` to enable scalable vector forms of the `RAX1` instruction.
- `+sve2-sm4` to enable scalable vector forms of `SM4E` and `SM4EKEY` instructions.

You can use one or more of these options. Each option also implies `+sve2`. For example, `+sve2-aes +sve2-bitperm+sve2-sha3+sve2-sm4` enables all base and optional instructions. For clarity, you can include `+sve2` if necessary.

See -march and -mcpu in the *Arm Compiler for Embedded FuSa Reference Guide* for more information.

Example - Naïve step-1 daxpy

daxpy is a subroutine of the Basic Linear Algebra Subroutines (BLAS) that operates on two arrays of double-precision floating-point numbers. A slice is taken of each of these arrays. For each element in these slices, an element (`x`) in the first array is multiplied by a constant (`a`), then added to the element (`y`) from the second array. The result is stored back to the second array at the same index.

This example presents a step-1 *daxpy* implementation, where the indices of `x` and `y` start at 0 and increment by 1 for each iteration. A C code implementation might look like the following:

```
void daxpy_1_1(int64_t n, double da, double *dx, double *dy)
{
    for (int64_t i = 0; i < n; ++i) {
        dy[i] = dx[i] * da + dy[i];
    }
}
```

Here is an ACLE equivalent:

```
void daxpy_1_1(int64_t n, double da, double *dx, double *dy)
{
    int64_t i = 0;
    svbool_t pg = svwhilelt_b64(i, n);                              // [1]
    do {
        svfloat64_t dx_vec = svld1(pg, &dx[i]);                     // [2]
        svfloat64_t dy_vec = svld1(pg, &dy[i]);                     // [2]
        svst1(pg, &dy[i], svmla_x(pg, dy_vec, dx_vec, da));         // [3]
        i += svcntd();                                              // [4]
        pg = svwhilelt_b64(i, n);                                   // [1]
    }
    while (svptest_any(svptrue_b64(), pg));                         // [5]
}
```

Example notes

[1] - Initialize a predicate register to control the loop. `_b64` specifies a predicate for 64-bit elements. Conceptually, this operation creates an integer vector starting at `i` and incrementing by 1 in each subsequent lane. The predicate lane is active if this value is less

than `n`. Therefore, this loop is safe, if inefficient, even if `n` ≤ 0. The same operation is used at the bottom of the loop, to update the predicate for the next iteration.

[2] - Load some values into an SVE vector, which is guarded by the loop predicate. Lanes where this predicate is false do not perform any load (and so do not generate a fault), and set the result value to 0.0. The number of lanes that are loaded depends on the vector width, which is only known at runtime.

[3] - Perform a floating-point multiply-add operation, and pass the result to a store. The `_x` on the `MLA` indicates we do not care about the result for inactive lanes. This gives the compiler maximum flexibility in choosing the most efficient instruction. The result of this operation is stored at address `&dy[i]`, guarded by the loop predicate. Lanes where the predicate is false are not stored, and the value in memory retains its prior value.

[4] - Increment `i` by the number of double-precision lanes in the vector.

[5] - `ptest` returns true if any lane of the (newly updated) predicate is active, which causes control to return to the start of the while loop if there is any work left to do.

Ideal assembler output:

```
daxpy_1_1:
    MOV Z2.D, D0            // da
    MOV X3, #0              // i
    WHILELT P0.D, X3, X0    // i, n
loop:
    LD1D Z1.D, P0/Z, [X1, X3, LSL #3]
    LD1D Z0.D, P0/Z, [X2, X3, LSL #3]
    FMLA Z0.D, P0/M, Z1.D, Z2.D
    ST1D Z0.D, P0, [X2, X3, LSL #3]
    INCD X3                 // i
    WHILELT P0.D, X3, X0    // i, n
    B.ANY loop
    RET
```

Example - Naïve general daxpy

This example presents a general *daxpy* implementation, where the indices of `x` and `y` start at 0 and are then incremented by unknown (but loop-invariant) strides each iteration.

```
void daxpy(int64_t n, double da, double *dx, int64_t incx,
           double *dy, int64_t incy)
{
  svint64_t incx_vec = svindex_s64(0, incx);                                    // [1]
  svint64_t incy_vec = svindex_s64(0, incy);                                    // [1]
  int64_t i = 0;
  svbool_t pg = svwhilelt_b64(i, n);                                            // [2]
  do {
      svfloat64_t dx_vec = svld1_gather_index(pg, dx, incx_vec);                // [3]
      svfloat64_t dy_vec = svld1_gather_index(pg, dy, incy_vec);                // [3]
      svst1_scatter_index(pg, dy, incy_vec, svmla_x(pg, dy_vec, dx_vec, da)); // [4]
      dx += incx * svcntd();                                                    // [5]
      dy += incy * svcntd();                                                    // [5]
      i += svcntd();                                                            // [6]
      pg = svwhilelt_b64(i, n);                                                 // [2]
  }
  while (svptest_any(svptrue_b64(), pg));                                       // [7]
```

```
}
```

Example notes

[1] - For each of `x` and `y`, initialize a vector of indices, starting at 0 for the first lane and incrementing by `incx` and `incy` respectively in each subsequent lane.

[2] - Initialize or update the loop predicate.

[3] - Load a vector's worth of values, which are guarded by the loop predicate. Lanes where this predicate is false do not perform any load (and so do not generate a fault), and set the result value to 0.0. This time, a base + vector-of-indices gather load, is used to load the required non-consecutive values.

[4] - Perform a floating-point multiply-add operation, and pass the result to a store. This time, the base + vector-of-indices scatter store is used to store each result in the correct index of the `dy[]` array.

[5] - Instead of using `i` to calculate the load address, increment the base pointer, by multiplying the vector length by the stride.

[6] - Increment `i` by the number of double-precision lanes in the vector.

[7] - Test the loop predicate to work out whether there is any more work to do, and loop back if appropriate.

8. Alignment support in Arm Compiler for Embedded FuSa 6

Arm® Compiler for Embedded FuSa 6 provides control over some aspects of alignment through options, keywords, and attributes.

When a processor accesses instructions and data, the access is either aligned or unaligned. An access is aligned if the address is a multiple of the element size. Otherwise, the access is unaligned. The element size depends on the processor architecture and the data type, such as `char` and `int`. For types such as structures, the alignment might be more complicated depending on the type of each structure member.

We can consider alignment as two distinct aspects, instruction alignment and data access alignment.

Instruction alignment

Instructions in the Arm architecture are aligned as follows:

- A32 and A64 instructions are word-aligned.
- T32 and ThumbEE instructions are halfword-aligned.
- Java bytecodes are byte-aligned.

Instruction alignment is defined as a power of 2. That is, an address `a` is 2^n byte aligned only if it is a multiple of 2^n.

Any attempt to fetch an instruction from a misaligned location results in a PC alignment fault.

Data access alignment

When the memory address of a data item is a multiple of the element size, then the data has natural alignment. A processor accesses memory most efficiently when the data has natural alignment. Sometimes, it might be necessary to insert padding to ensure the natural alignment of data.

For a variable `x` with a basic type of size `n` bytes, such as `int`, then `x` is aligned only if `x` is placed at an `n`-byte aligned address. However, the size of more complex types does not contribute to the alignment in the same way as basic data types.

For a complex data type, such as a structure, the alignment is that of the member with the biggest alignment. Also, if all members in a structure are aligned, then the structure is aligned.

The following table shows the natural alignment requirement for some basic data types:

Table 8-1: Armv8 AArch32 alignment requirements of load and store instructions

Type	Size in bytes (bits)	Natural alignment requirement
`char`	1 (8)	Address divisible by 1 - Always aligned

Type	Size in bytes (bits)	Natural alignment requirement
`int`	4 (32)	Address divisible by 4
`long`	8 (64)	Address divisible by 8
`short`	2 (16)	Address divisible by 2 - Even addresses

In practice, data might not always be aligned. You can override the natural alignment of a variable in your source code with attributes or keywords, such as the `__attribute__((aligned))` variable attribute. Overriding the alignment can ultimately cause the compiler to generate code with unaligned accesses through attributes such as `__attribute__((packed))` or using unsafe cast alignment. However, unaligned accesses might cause alignment faults. Your code might or might not execute without fault depending on:

- Whether the processor supports unaligned accesses.
- Whether the instruction generated supports unaligned accesses. For example, `LDRD` does not support unaligned accesses and generates a run-time exception if it attempts to access unaligned data.

If natural alignment is the most efficient way that a processor can access data, why change it? Using a custom alignment can significantly improve performance or save memory, especially with structures.

Arm architectures support two types of memory:

Normal memory

Normal memory is regular memory for code, data, heap, and stack. This memory has the following properties:

- It can contain executable code.
- It can be cached.
- It can be reordered.
- Speculative load is allowed.
- The memory can be buffered.
- Unaligned access might be supported based on other settings such as the SCTLR.A bit in AArch32 state or the SCTLR_ELx.nAA bit in AArch64 state.

Device memory

Device memory is a memory-mapped I/O register region. This memory has the following properties:

- It is never cached.
- It is not executable.
- No instruction fetch occurs.
- No speculative data access occurs.
- Writes can be buffered.
- All accesses must be aligned.

Any unaligned access to Device memory generates an Alignment fault.

For more information on Normal and Device memory and restrictions for each supported architecture, see:

- ARMv6-M Architecture Reference Manual.
- ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition.
- ARMv7-M Architecture Reference Manual.
- Arm Architecture Reference Manual for A-profile architecture, for Armv8-A and Armv9-A architectures.
- Armv8-M Architecture Reference Manual.

Alignment, architectures, and performance

Different Arm architectures might or might not support aligned accesses.

For example, Arm® Cortex®-M0 does not support unaligned accesses. Therefore, if some instructions complete a transaction of a piece of data that does not lie on a word boundary, then the processor throws an Alignment fault at the execution level.

Some processors support architectures such as Armv7 that allow for unaligned accesses. Therefore, if some instructions load data from memory, and this data does not lie on a word boundary, the processor still completes the translation. However, these unaligned accesses have a cost.

For example, data might begin at an address that is not divisable by 4, such as at address `0x1001`. In this case, the processor must first access the data at address `0x1000` and then apply an algorithm to access the required data value at byte `0x1001`. This operation takes time and lowers performance. Therefore, having all data addresses aligned is more efficient. Data that spans page boundaries and caching can also increase the number of transactions, and degrade performance as a result.

8.1 Aligned and unaligned accesses

A memory access is aligned when the data being accessed is `n` bytes long and the datum address is `n`-byte aligned. That is, the address used in the memory access is divisible by the size of the data being fetched.

Access alignment relates to the lower level on the software stack, rather than being present at the source-code level. Access alignment concerns memory transactions that are performed at an instruction level that might be part of a more complex piece of data.

For example, an attempt to fetch a complex type from memory, such as a `struct` at C level that contains `char` and `short` types might occur in multiple load instructions. The alignment of accesses is dictated by checking whether each load instruction is aligned. A load is aligned when the address of the load after applying offsets is divisible by the size of the load being fetched. That is, checking whether the natural word boundaries are honored.

Table 8-2: Access alignment for variants of load instructions

Load variant	Size of load	The access is aligned if:	Supports unaligned access [1]
`LDR`	Word size (4 bytes, 32 bits)	The final address is a multiple of 4.	Yes
`LDRSH`	Half word size (2 bytes, 16 bits)	The final address is a multiple of 2.	Yes
`LDRB`	Byte size (1 byte, 8 bits)	The final address is a multiple of 1. The access is always aligned.	Yes
`LDRD`	Double word size (8 bytes, 64 bits)	The final address is a multiple of 8.	No - Generates a run-time exception.

Table note

[1] Assumes that the processor supports unaligned accesses.

Arm®v7 and later architectures must support unaligned data accesses for some load and store instructions.

Table 8-3: Armv8 AArch32 alignment requirements of load and store instructions

Instructions	Alignment check	Result if check fails when SCTLR.A or HSCTLR.A is 0	Result if check fails when SCTLR.A or HSCTLR.A is 1
`LDRB`, `LDREXB`, `LDRBT`, `LDRSB`, `LDRSBT`, `STRB`, `STREXB`, `STRBT`, `TBB`	None	-	-
`LDRH`, `LDRHT`, `LDRSH`, `LDRSHT`, `STRH`, `STRHT`, `TBH`	Halfword	Unaligned access	Alignment fault
`LDREXH`, `STREXH`, `LDAH`, `STLH`, `LDAEXH`, `STLEXH`	Halfword	Alignment fault	Alignment fault
`LDR`, `LDRT`, `STR`, `STRT` `PUSH`, encodings T3 and A2 only `POP`, encodings T3 and A2 only	Word	Unaligned access	Alignment fault
`LDREX`, `STREX`, `LDA`, `STL`, `LDAEX`, `STLEX`	Word	Alignment fault	Alignment fault
`LDREXD`, `STREXD`, `LDAEXD`, `STLEXD`	Doubleword	Alignment fault	Alignment fault
All forms of `LDM` and `STM`, `LDRD`, `RFE`, `SRS`, `STRD`	Word	Alignment fault	Alignment fault
`LDC`, `STC`	Word	Alignment fault	Alignment fault
`VLDM`, `VLDR`, `VPOP`, `VPUSH`, `VSTM`, `VSTR`	Word	Alignment fault	Alignment fault
`VLD1`, `VLD2`, `VLD3`, `VLD4`, `VST1`, `VST2`, `VST3`, `VST4`, all with standard alignment	Element size	Unaligned access	Alignment fault
`VLD1`, `VLD2`, `VLD3`, `VLD4`, `VST1`, `VST2`, `VST3`, `VST4`, all with `:<align>` specified [1]	As specified by `:<align>`	Alignment fault	Alignment fault

Table note

[1] The `:` character is the preferred separator, but `@<align>` is also supported.

Table 8-4: Armv7 alignment requirements of load and store instructions

Instructions	Alignment check	Result if check fails when SCTLR.A is 0	Result if check fails when SCTLR.A is 1
`LDRB`, `LDREXB`, `LDRBT`, `LDRSB`, `LDRSBT`, `STRB`, `STREXB`, `STRBT`, `SWPB`, `TBB`	None	-	-
`LDRH`, `LDRHT`, `LDRSH`, `LDRSHT`, `STRH`, `STRHT`, `TBH`	Halfword	Unaligned access	Alignment fault
`LDREXH`, `STREXH`	Halfword	Alignment fault	Alignment fault
`LDR`, `LDRT`, `STR`, `STRT` `PUSH`, encodings T3 and A2 only `POP`, encodings T3 and A2 only	Word	Unaligned access	Alignment fault
`LDREX`, `STREX`	Word	Alignment fault	Alignment fault
`LDREXD`, `STREXD`	Doubleword	Alignment fault	Alignment fault
All forms of `LDM` and `STM`, `LDRD`, `RFE`, `SRS`, `STRD`, `SWP` `PUSH`, except for encodings T3 and A2 `POP`, except for encodings T3 and A2	Word	Alignment fault	Alignment fault
`LDC`, `LDC2`, `STC`, `STC2`	Word	Alignment fault	Alignment fault
`VLDM`, `VLDR`, `VPOP`, `VPUSH`, `VSTM`, `VSTR`	Word	Alignment fault	Alignment fault
`VLD1`, `VLD2`, `VLD3`, `VLD4`, `VST1`, `VST2`, `VST3`, `VST4`, all with standard alignment [1]	Element size	Unaligned access	Alignment fault
`VLD1`, `VLD2`, `VLD3`, `VLD4`, `VST1`, `VST2`, `VST3`, `VST4`, all with `:<align>` specified [2]	As specified by `:<align>`	Alignment fault	Alignment fault

Table notes

[1] These element and structure load and store instructions are only in the Advanced SIMD Extension to the A32 and T32 instruction sets.

[2] The `:` character is the preferred separator, but `@<align>` is also supported.

8.2 Unaligned access support in Arm Compiler for Embedded FuSa

The Arm®v6 architecture, with the exception of Armv6-M, introduced the first hardware support for unaligned accesses. Cortex®-A and Cortex-R processors can deal with unaligned accesses in hardware, removing the need for software routines.

Support for unaligned accesses is limited to a subset of load and store instructions:

- `LDRB`, `LDRSB`, and `STRB`.
- `LDRH`, `LDRSH`, and `STRH`.
- `LDR` and `STR`.

Instructions that do not support unaligned accesses include:

- `LDM` and `STM`.
- `LDRD` and `STRD`.

Also, unaligned accesses are only allowed to regions marked as Normal memory type. To enable unaligned access support, set the SCTLR.A bit in the system control coprocessor. Attempts to perform unaligned accesses when not allowed cause an Alignment fault, which is taken as a Data Abort exception. See Unaligned data access for more information.

How hardware supports unaligned accesses

In many cases, a processor cannot generate an unaligned access on its interfaces to the memory system. This situation applies to caches, Tightly Coupled Memories (TCMs), and the system bus. In these cases, the processor generates a series of accesses to implement the unaligned access. This method is similar to the software routines used for earlier processors.

For example:

```
MOV r1, #0x8001
LDR r0, [r1]
```

Most modern Arm processors have 64-bit or 128-bit interfaces. In this example, a processor typically reads the 64-bit or 128-bit block containing bytes `0x8001`, `0x8002`, `0x8003`, and `0x8004`. The processor discards the other bytes.

For another example:

```
MOV r1, #0x81FC
LDR r0, [r1]
```

The four bytes of this load span both a 64-bit and 128-bit boundary. Therefore, with either interface width, the processor has to perform two reads.

In both of these examples, it is possible to see that unaligned accesses require more work by the hardware. While more efficient than the software routines required by previous processors, it is still less efficient than aligned accesses.

Pointer alignment in C

When compiling C, variables are by default architecturally aligned. A global of type `int` or `uint32_t` is 4-byte aligned in memory. Similarly, a pointer of type `int*` is expected to contain a 4-byte aligned address.

Where this is not the case, or might not be the case, the variable or pointer must be marked with the `__unaligned` keyword. This keyword is a warning to the compiler that the variable or pointer is potentially unaligned. That is, it reduces the expected alignment of the pointer to 1-byte. For more information, see __unaligned.

For a structure layout, you must use the `__attribute__((packed))` variable or type attribute to ensure the smallest possible alignment of structure members. For more information, see:

- __attribute__((packed)) type attribute.
- __attribute__((packed)) variable attribute.

Compiler assumptions

When compiling for an Armv7-A or Armv7-R processor, Arm Compiler for Embedded FuSa assumes that it can use unaligned accesses.

The `-mno-unaligned-access` option tells the compiler not to knowingly generate unaligned accesses. What is the significance of knowingly?

As mentioned in the previous section, a pointer must contain an address with correct alignment for the type:

- `uint32_t*` requires 4-byte alignment.
- `uint16_t*` requires 2-byte alignment.
- `uint8_t*` requires 1-byte alignment.

For structures, the alignment is that of the most aligned member.

The compiler generates code on the assumption that a pointer is correctly aligned. It does not add code to perform run-time checks. A pointer might contain an incorrectly aligned address for many reasons. A common cause is casting, for example:

```
uint8_t tmp;
uint32_t* pMyPointer = (uint32_t*)(&tmp);
```

This code takes the address of a `uint8_t` variable, then casts that address as a `uint32_t` pointer. The compiler still assumes that `pMyPointer` is correctly aligned for a `uint32_t` pointer. The compiler might then unknowingly generate code that results in an unaligned access.

You can avoid this situation with the `__unaligned` qualifier, for example:

```
uint8_t tmp;
__unaligned uint32_t* pMyPointer = (__unaligned uint32_t*)(&tmp);
```

Code Generation

When unaligned accesses are permitted, the compiler continues to use instructions that support unaligned accesses for accesses through `__unaligned` pointers. For example `LDR` and `STR` instructions. However, it does not use instructions that do not support unaligned accesses, such as `LDM`.

When unaligned accesses are not permitted, because you specified the compiler option `-mno-unaligned-access`, the compiler accesses `__unaligned` data by performing a number of aligned accesses. Usually, this access is done by calling a library function such as `__aeabi_uread4()`.

Device Memory

Address regions that access peripherals rather than memory must be marked as Device memory. Depending on the processor, this memory might be configured in the Memory Protection Unit (MPU) or the Memory Management Unit (MMU). Unaligned accesses are not permitted to these regions even when unaligned access support is enabled. If an unaligned access is attempted, the processor generates an Alignment fault.

The compiler does not have any information about which address ranges are Device memory. Therefore, it is your responsibility to ensure the alignment of accesses to devices. In practice, peripheral registers are usually at aligned addresses. It is also usual to access peripheral registers through `volatile` variables or pointers. Use of `volatile` restricts the compiler to accessing the data with the size of access specified where possible. For more information on the restrictions imposed on `volatile` types, see the *Volatile Data Types* section of the Procedure Call Standard for the Arm Architecture.

It is also necessary to avoid using C library functions such as `memcpy()` to access Device memory, because there is no guarantee of the type of accesses these functions use. If it is necessary to copy a buffer of memory to a Device memory, you must provide a suitable copying routine and call this routine instead of `memcpy()`.

For more information, see Device and Strongly-ordered memory.

Performance

If code frequently accesses unaligned data, there might be a performance advantage to enabling unaligned accesses. However, the extent of this advantage depends on many factors. Even though this support allows a single instruction to access unaligned data, it often requires multiple bus accesses to occur. Therefore, the bus transactions performed by an unaligned access might be similar to those performed by the multiple instructions used when unaligned access support is disabled. The code without unaligned access support has to perform various shift and logical operations. However, on a multi-issue processor the execution time of these operations might be hidden by executing them in parallel with the memory accesses. There is also a function call overhead when using functions such as `__aeabi_uread4()`, though branch prediction might reduce the impact of using these functions.

Related information

-munaligned-access, -mno-unaligned-access
__unaligned
Volatile variables
How can I debug an Arm AArch64 Alignment Abort?
memcpy and memset unaligned access and alignment fault

8.3 Alignment at the source code and compilation level

On modern processors, how a compiler places data in the final binary depends on alignment considerations to generate optimized code.

How a C compiler places basic C data types in memory is not arbitrary. Data does not normally start at arbitrary byte addresses in memory. Rather, each type except `char` has an alignment requirement:

- A single-byte `char` can start on any byte address.
- A 2-byte `short` must start on an even address.
- A 4-byte `int` or `float` must start on an address divisible by 4.
- An 8-byte `long` or `double` must start on an address divisible by 8.

Whether the data is signed or unsigned makes no difference.

That is, basic C types on a standard Instruction Set Architecture (ISA) are self-aligned. Pointers, whether 32-bit (4-byte) or 64-bit (8-byte) are also self-aligned.

Self-alignment makes access faster because it facilitates generating single-instruction fetches and puts of the typed data. However, without alignment constraints, the code might perform two or more accesses that span machine-word boundaries. Characters are a special case and they are equally expensive whereever they live inside a single machine word. That is why they do not have a preferred alignment.

To ensure natural alignment, it might be necessary to insert some padding between structure elements or after the last element of a structure.

Example: Padding between structure elements

This example shows how you can reduce padding by modifying your source code.

The example is available in Example of padding between structure elements.

8.4 Example of padding between structure elements

This example shows how you can reduce padding by modifying your source code.

For more information, see Alignment at the source code and compilation level.

Example: Remove padding by modifying the structure layout in source code

You might have the following structure:

```
typedef struct
{
    char a;
    int b;
    char c
    short d;
} my_struct_t;
```

After compiling, the layout in memory is determined by the `int` type, because that has the highest alignment. For example:


Note

For this example and the following examples, the most important part of the address is the last two hexadecimal values. Therefore, `??????` means any address where the data might be placed by the compiler.

Address	Byte 0	Byte 1	Byte 2	Byte 3
0x??????00	...	...	...	...
0x??????04	char a	padding	padding	padding
0x??????08	int b	int b	int b	int b
0x??????0c	char c	padding	short d	short d
0x??????10	...	...	...	...

However, by changing the layout of the structure in the source code, you can assist the compilation and reduce or remove the padding. For this example, change the `struct` to:

```
typedef struct
{
    char a;
    char c
    short d;
    int b;
} my_struct_t;
```

Placing the `int` after the `short` removes the padding:

Address	Byte 0	Byte 1	Byte 2	Byte 3
0x??????00	...	...	...	...
0x??????04	char a	char c	short d	short d

Address	Byte 0	Byte 1	Byte 2	Byte 3
0x??????08	int b	int b	int b	int b
0x??????0c	...	...	...	...
0x??????10	...	...	...	...

Example: A struct that cannot have padding removed

You might have the following structure:

```
typedef struct
{
    char a;
    short d;
    int b;
} my_struct_t;
```

After compiling, the layout in memory is, for example:

Address	Byte 0	Byte 1	Byte 2	Byte 3
0x??????00	...	...	...	...
0x??????04	char a	padding	short d	short d
0x??????08	int b	int b	int b	int b
0x??????0c	...	...	...	...
0x??????10	...	...	...	...

As a consequence, not only is the data aligned in memory, but all accesses and all generated instructions are aligned.

Example: Alignment of instructions

Create the file `main.c` containing the following C code:

```
#include <stdio.h>

struct my_struct
{
    char a;
    short b;
    int c;
};

struct my_struct f;

int main(void)
{
    printf("%d\n", f.a + f.b + f.c);

    return 0;
}
```

To compile the program, enter:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -O1 -Wcast-align -c main.c -o
 main.o
```

To view the disassembly and symbol table with `fromelf`, enter:

```
fromelf -cdef -s main.o
```

The output shows:

```
...
    main
        0x00000000:    b580        ..      PUSH     {r7,lr}
        0x00000002:    f2400000    @...    MOVW     r0,#:LOWER16: f
        0x00000006:    f2c00000    ....    MOVT     r0,#:UPPER16: f
        0x0000000a:    7801        .x      LDRB     r1,[r0,#0]
        0x0000000c:    f9b02002    ...     LDRSH    r2,[r0,#2]
        0x00000010:    6840        @h      LDR      r0,[r0,#4]
        0x00000012:    4411        .D      ADD      r1,r1,r2
        0x00000014:    4401        .D      ADD      r1,r1,r0
        0x00000016:    a002        ..      ADR      r0,{pc}+0xa ; 0x20
        0x00000018:    f7fffffe    ....    BL       __2printf
        0x0000001c:    2000        .       MOVS     r0,#0
        0x0000001e:    bd80        ..      POP      {r7,pc}
...
      #  Symbol Name                 Value      Bind  Sec  Type  Vis  Size
    ============================================================================
...
      7  f                           0x00000000   Gb    7  Data  Hi   0x8
...
```

After working out the initial value for register r0, it is possible to conclude that the various fetching operations in this example represent aligned accesses.

However, if space is a constraint you can force the compiler to overlook the alignment requirements to save space. Arm® Compiler for Embedded FuSa 6 provides this feature with the `__attribute__((packed))` type attribute. For more information, see __attribute__((packed)) type attribute.

Modify the `struct` in `main.c` as follows:

```
#include <stdio.h>

struct __attribute__((packed)) my_struct
{
    char a;
    short b;
    int c;
};
```

The layout in memory is now:

Address	Byte 0	Byte 1	Byte 2	Byte 3
0x??????00	...	...	...	...
0x??????04	char a	short d	short d	int b
0x??????08	int b	int b	int b	...
0x??????0c	...	...	...	...
0x??????10	...	...	...	...

View the contents of the object file using the `fromelf` command:

```
...
    main
        0x00000000:    b580        ..      PUSH     {r7,lr}
        0x00000002:    f2400000    @...    MOVW     r0,#:LOWER16: f
        0x00000006:    f2c00000    ....    MOVT     r0,#:UPPER16: f
        0x0000000a:    7801        .x      LDRB     r1,[r0,#0]
        0x0000000c:    f9b02001    ...     LDRSH    r2,[r0,#1]
        0x00000010:    f8d00003    ....    LDR      r0,[r0,#3]
        0x00000014:    4411        .D      ADD      r1,r1,r2
        0x00000016:    4401        .D      ADD      r1,r1,r0
        0x00000018:    a002        ..      ADR      r0,{pc}+0xc ; 0x24
        0x0000001a:    f7fffffe    ....    BL       __2printf
        0x0000001e:    2000        .       MOVS     r0,#0
        0x00000020:    bd80        ..      POP      {r7,pc}
        0x00000022:    bf00        ..      NOP
...
      #  Symbol Name                 Value      Bind  Sec  Type  Vis  Size
    ================================================================================
...
      7  f                           0x00000000   Gb    7  Data  Hi   0x7
...
```

You can see that the size of `f` has changed to 7 bytes rather than 8 in the unpacked version.

Also, assuming that r0 contains an aligned address, then:

- `LDRB r1,[r0,#0]` is an aligned access when fetching the `char`.
- `LDRSH r2,[r0,#1]` is an unaligned access when fetching the `short`.
- `LDR r0,[r0,#3]` is an unaligned address when fetching the `int`.

Although this example shows that `f` is unaligned in memory, you can force the compiler to perform aligned accesses to the elements of `f` using the command-line option `-mno-unaligned-access`.

Compile `main.c` again with `-mno-unaligned-access`:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -O1 -Wcast-align -mno-unaligned-
access -c main.c -o main.o
```

View the contents of the object file using the `fromelf` command:

```
...
    main
        0x00000000:    b580        ..      PUSH     {r7,lr}
        0x00000002:    f2400000    @...    MOVW     r0,#:LOWER16: f
        0x00000006:    f2c00000    ....    MOVT     r0,#:UPPER16: f
        0x0000000a:    f9901002    ....    LDRSB    r1,[r0,#2]
        0x0000000e:    7843        Cx      LDRB     r3,[r0,#1]
        0x00000010:    7802        .x      LDRB     r2,[r0,#0]
        0x00000012:    f890c004    ....    LDRB     r12,[r0,#4]
        0x00000016:    ea432101    C..!    ORR      r1,r3,r1,LSL #8
        0x0000001a:    f8103f03    ...?    LDRB     r3,[r0,#3]!
        0x0000001e:    4411        .D      ADD      r1,r1,r2
        0x00000020:    7882        .x      LDRB     r2,[r0,#2]
        0x00000022:    78c0        .x      LDRB     r0,[r0,#3]
        0x00000024:    ea43230c    C..#    ORR      r3,r3,r12,LSL #8
        0x00000028:    ea422000    B..     ORR      r0,r2,r0,LSL #8
        0x0000002c:    ea434000    C..@    ORR      r0,r3,r0,LSL #16
```

```
        0x00000030:    4401        .D      ADD      r1,r1,r0
        0x00000032:    a002        ..      ADR      r0,{pc}+0xa ; 0x3c
        0x00000034:    f7fffffe    ....    BL       __2printf
        0x00000038:    2000        .       MOVS     r0,#0
        0x0000003a:    bd80        ..      POP      {r7,pc}
...
      #  Symbol Name                 Value      Bind  Sec  Type  Vis  Size
    ================================================================================
...
      7  f                           0x00000000   Gb    7  Data  Hi   0x7
...
```

You can see that `f` is still 7 bytes and is unaligned.

However all the accesses performed are aligned, which is possible to see because they are all byte accesses (`LDRB` and `LDRSB`).

Therefore, the code occupies the same space but relies on aligned accesses. Although the aligned accesses are useful for performance reasons, other factors that are out of the control of the compiler might degrade the peformance. For example, accesses across page boundaries and caching.

8.5 Alignment and unsafe casting

For some cases, such as unsafe casting, the `armclang` option `-mno-unaligned-access` might not have the effect you expect.

For example, unsafe casting is when you initialize a variable of one data type, and then cast it to another data type with a bigger alignment requirement.

If you add the `-mno-unaligned-access` option during compilation, unaligned accesses still happen at the assembly level.

Example: Casting a char pointer to an int pointer

This example shows the result of casting a `char` pointer to an `int` pointer.

The example is available in Example of casting a char pointer to an int pointer.

Related information

-munaligned-access, -mno-unaligned-access
--unaligned_access, --no_unaligned_access

8.6 Example of casting a char pointer to an int pointer

This example shows the result of casting a `char` pointer to an `int` pointer.

For more information, see Alignment and unsafe casting.

Example: Unsafe casting of a char pointer to an int pointer

Create the file `unsafe_cast.c` containing the following C code:

```
#include <stdio.h>

char *c = (char *)0xc001;
int main(void)
{
    int *x = (int *)c;
    printf("%x\n", x[0]);
    return 0;
}
```

To compile the program, enter:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -O1 -c unsafe_cast.c -o
 unsafe_cast.o
```

To view the disassembly, enter:

```
fromelf -cdef -s unsafe_cast.o
```

The output shows:

```
...
  main
      0x00000000:    b580        ..      PUSH     {r7,lr}
      0x00000002:    f2400000    @...    MOVW     r0,#:LOWER16: c
      0x00000006:    f2c00000    ....    MOVT     r0,#:UPPER16: c
      0x0000000a:    6800        .h      LDR      r0,[r0,#0]
      0x0000000c:    6801        .h      LDR      r1,[r0,#0]
      0x0000000e:    a002        ..      ADR      r0,{pc}+0xa ; 0x18
      0x00000010:    f7fffffe    ....    BL       __2printf
      0x00000014:    2000        .       MOVS     r0,#0
      0x00000016:    bd80        ..      POP      {r7,pc}
...
```

This example shows the following:

- The `MOVW` and `MOVT` instructions load r0 with the address of `c`.
- `LDR r0,[r0,#0]` then loads the contents of `c` into r0, which from the source code is `0xc001`.
- `LDR r1,[r0,#0]` loads the contents of address `0xc001` into r1.

The last `LDR` instruction is an unaligned access because it fetches a 4-byte integer starting at address `0xc001` which means that the word boundaries are crossed:

Address	Byte 0	Byte 1	Byte 2	Byte 3
0xc000	...	c	c	c
0xc004	c	...	...	...

Compiling `unsafe_cast.c` with `-mno-unaligned-access` still generates the same assembly.

You can detect unsafe casts with the `-Wcast-align` compiler option, for example:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -O1 -Wcast-align -c
 unsafe_cast.c -o unsafe_cast.o
unsafe_cast.c:6:14: warning: cast from 'char *' to 'int *' increases required
 alignment from 1 to 4 [-Wcast-align]
    6 |     int *x = (int *)c;
      |              ^~~~~~~~
1 warning generated.
```

To abort the compilation when this situation occurs, use the `-Werror=cast-align` compiler option, for example:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -O1 -Werror=cast-align -c
 unsafe_cast.c -o unsafe_cast.o
unsafe_cast.c:6:14: error: cast from 'char *' to 'int *' increases required
 alignment from 1 to 4 [-Werror,-Wcast-align]
    6 |     int *x = (int *)c;
      |              ^~~~~~~~
1 error generated.
```

Although we initially have unaligned accesses, the code can still run on a processor that allows unaligned accesses. However, some instructions such as `LDRD` only allow for aligned accesses. Therefore, providing an unaligned address to `LDRD` causes a fault. In most cases, the compiler ensures that `LDRD` instructions always work with aligned addresses. The only situation where it does not follows from unsafe pointer casting.

Example: Unsafe pointer casting

Create the file `init_pointers.c` containing the following C code:

```
#include <stdio.h>

char *c = (char *)0xc001;
void init_pointers(void)
{
    *c = 0xaa;
}

int main(void)
{
    int *x = (int *)c;
    init_pointers();
    printf("%x, %x\n", x[0], x[1]);
    return 0;
}
```

To compile the program, enter:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -O1 -c init_pointers.c -o
 init_pointers.o
```

To view the disassembly, enter:

```
fromelf -cdef -s init_pointers.o
```

The output shows:

```
...
    main
        0x00000000:    b580        ..      PUSH     {r7,lr}
        0x00000002:    f2400000    @...    MOVW     r0,#:LOWER16: c
        0x00000006:    f2c00000    ....    MOVT     r0,#:UPPER16: c
        0x0000000a:    6800        .h      LDR      r0,[r0,#0]
        0x0000000c:    21aa        .!      MOVS     r1,#0xaa
        0x0000000e:    7001        .p      STRB     r1,[r0,#0]
        0x00000010:    e9d01200    ....    LDRD     r1,r2,[r0,#0]
        0x00000014:    a002        ..      ADR      r0,{pc}+0xc ; 0x20
        0x00000016:    f7fffffe    ....    BL       __2printf
        0x0000001a:    2000        .       MOVS     r0,#0
        0x0000001c:    bd80        ..      POP      {r7,pc}
        0x0000001e:    bf00        ..      NOP
```

You can still use the `-Wcast-align` and `-Werror=cast-align` compiler options to detect these situations.

8.7 Instruction alignment of functions and loops

Alignment of functions and loops is commonly used in program performance optimization.


Note

This topic includes descriptions of [COMMUNITY] features. See Support level definitions.

The following example shows a simple loop in assembly code:

```
    .globl ASMDELAY
ASMDELAY:
    subs r0,r0,#1
    bne ASMDELAY
    bx lr
```

To simulate different parameters of loop alignment, you can insert padding on top of the code. However, the performance difference depends on the padding that you insert.

In application code that you are developing, it is harder to quantify and qualify the effects of the alignment on performance. The microarchitecture and cache interactions with your application can influence the effects.

For example, a bigger alignment boosts execution performance in general. However, loops usually rely on the repeated code. Therefore, if the alignment is too big, the code might occupy more memory than necessary and might not fit in cache. This situation hinders the performance. Quantifying this trade-off is difficult, because making alignment decisions is difficult.

In general, you can try to set loop and function alignment to coincide to cache line size.

Processor caches transfer data from and to main memory in chunks called cache lines. A typical size for the cache line size is 64 bytes. Using an alignment larger than 64 bytes means crossing cache line boundaries that result in more fetches.

However, a larger cache line size could mean that data has enough space to be properly aligned, and in general, executing the code would be faster.

Alternatively, a smaller alignment than the cache line size might produce faster code because it increases the use of the cache. However, to fit in these space boundaries it might also mean that data must be unaligned, therefore, lowering performance.

The following [COMMUNITY] command-line options allow you to regulate the alignment of functions and loops with:

- `-falign-functions`.
- `-falign-loops`.

For more information about these options, see the Clang command line argument reference.

8.8 Alignment and linking

Arm® Compiler for Embedded FuSa 6 supports the `armlink` alignment option `--no_unaligned_access`. This option checks whether all files being linked are compiled with the `-mno-unaligned-access` option.

Example: Linking with the --no_unaligned_access option

Create the file `main.c` containing the following C code:

```
#include <stdio.h>
#include "struct_packed.c"

struct my_struct f;

int main(void)
{
    printf("%d\n", f.a + f.b + f.c);

    return 0;
}
```

Create the file `struct_packed.c` containing the code:

```
struct __attribute__((packed)) my_struct
{
    char a;
    short b;
    int c;
};
```

To compile the files, enter:

```
armclang -mcpu=cortex-m3 --target=arm-arm-none-eabi -O1 main.c -c -o main.o
armclang -mcpu=cortex-m3 --target=arm-arm-none-eabi -O1 -mno-unaligned-access
 struct_packed.c -c -o struct_packed.o
```

To link the object files, enter:

```
armlink --no_unaligned_access main.o struct_packed.o -o alignment_link_example.axf
```

The link generates the following error:

```
Error: L6366E: main.o attributes are not compatible with the provided attributes .
Object main.o contains Build Attributes that are incompatible with the provided
 attributes.
  Tag_CPU_unaligned_access = The producer was permitted to generate architecture v6-
style unaligned data accesses (=1)
Finished: 2 information, 0 warning and 1 error messages.
```

If you add the `-mno-unaligned-access` option when compiling `main.c`, this error is not generated.

Related information

-munaligned-access, -mno-unaligned-access
--unaligned_access, --no_unaligned_access

9. Building for different target architectures

Arm® Compiler for Embedded FuSa allows you to build applications for various targets. You can build an application for a specific architecture or processor, and for specific features that are supported by the architecture or processor.

9.1 How to build for an Armv8-R AArch64 target without hardware floating-point support

This task shows you how to build an application for an Arm®v8-R AArch64 target without hardware floating-point support.

About this task

To build an application for an Armv8-R AArch64 target without hardware floating-point support, you must:

- Compile with an `-march` or `-mcpu` option for Armv8-R AArch64 that specifies the `+nofp` feature. For example, either `-march=armv8-r+nofp` or `-mcpu=cortex-r82+nofp`.
- Compile with `-mabi=aapcs-soft`.
- Link with `--cpu=8-R.64 --fpu=SoftVFP`.


Note

If your application includes assembly code, assembling with `+nofp` reports an error if your assembly code contains floating-point instructions. Therefore, we recommend that you assemble with both `+nofp` and `-mabi=aapcs-soft`.

Procedure

1. Create the file `main.c` containiing the following C code:

```
#include <stdio.h>
#include <math.h>

__attribute__((noinline)) void test_nofp(float a, float b)
{
    printf("%1.1f + %1.1f = %1.f\n", a, b, a + b);
    printf("floorf(%1.1f) = %1.f\n", a, floorf(a));
    printf("floorf(%1.1f) = %1.f\n", b, floorf(b));
}

int main(void)
{
    puts("Hello, world!");

    test_nofp(2.7f, -2.3f);

    return 0;
}
```

2. Compile and link with the following command:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-r+nofp -mabi=aapcs-soft -O1
 -Wl,--cpu=8-R.64 -Wl,--fpu=SoftVFP main.c -o aarch64-r.axf
```

3. Run the image on a suitable target. The image displays:

```
Hello, world!
2.7 + -2.3 = 0.4
floorf(2.7) = 2.0
floorf(-2.3) = -3.0
```

4. Run `fromelf` to display the disassembly:

```
fromelf --disassemble aarch64-r.axf
```

In the disassembly, you can see that no floating-point is used because:

- There are no FP registers for the `test_nofp()` function or `main()` function.
- There are no FP registers for the `floorf()` library function.

Related information

-mabi=<name> (armclang)
-march (armclang)
-mcpu (armclang)
--target
--cpu
--fpu
--disassemble

10. Mapping Code and Data to the Target

There are various options in Arm® Compiler for Embedded FuSa to control how code, data and other sections of the image are mapped to specific locations on the target.

10.1 What the linker does to create an image

The linker takes object files that a compiler or assembler produces and combines them into an executable image. The linker also uses a memory description to assign the input code and data from the object files to the required addresses in the image.

You can specify object files directly on the command line or specify a user library containing object files. The linker:

- Resolves symbolic references between the input object files.
- Extracts object modules from libraries to resolve otherwise unresolved symbolic references.
- Removes unused sections.
- Eliminates duplicate common groups and common code, data, and debug sections.
- Sorts input sections according to their attributes and names, and merges sections with similar attributes and names into contiguous chunks.
- Organizes object fragments into memory regions according to the grouping and placement information that is provided in a memory description.
- Assigns addresses to relocatable values.
- Generates either a partial object if requested, for input to another link step, or an executable image.

The linker has a built-in memory description that it uses by default. However, you can override this default memory description with command-line options or with a scatter file. The method that you use depends how much you want to control the placement of the various output sections in the image:

- Allow the linker to automatically place the output sections using the default memory map for the specified linking model. `armlink` uses default locations for the RO, RW, eXecute-Only (XO), and ZI output sections.
- Use the memory map related command-line options to specify the locations of the RO, RW, XO, and ZI output sections.
- Use a scatter file if you want to have the most control over where the linker places various parts of your image. For example, you can place individual functions at specific addresses or certain data structures at peripheral addresses.


Note

XO sections are supported only for images that are targeted at Arm®v6-M, Armv7-M, or Armv8-M architectures.

10.1.1 What you can control with a scatter file

A scatter file gives you the ability to control where the linker places different parts of your image for your particular target.

You can control:

- The location and size of various memory regions that are mapped to ROM, RAM, and FLASH.
- The location of individual functions and variables, and code from the Arm standard C and C++ libraries.
- The placement of sections that contain individual functions or variables, or code from the Arm standard C and C++ libraries.
- The priority ordering of memory areas for placing unassigned sections, to ensure that they get filled in a particular order.
- The location and size of empty regions of memory, such as memory to use for stack and heap.

If the location of some code or data lies outside all the regions that are specified in your scatter file, the linker attempts to create a load and execution region to contain that code or data.


Note

Multiple code and data sections cannot occupy the same area of memory, unless you place them in separate overlay regions.

10.1.2 Interaction of OVERLAY and PROTECTED attributes with armlink merge options

The `OVERLAY` and `PROTECTED` scatter-loading attributes modify the behavior of the `armlink` options `--merge` and `--merge_litpools`.

The following table describes how the `OVERLAY` and `PROTECTED` scatter-loading attributes affect the `armlink` options `--merge` and `--merge_litpools`. The terms const string and const value have the following meanings:

const string

A string literal from an ELF section with the `SHF_MERGE` and `SHF_STRINGS` flags.

const value

A constant defined in a constant pool where the constant pool is in the same section as the code that uses it.

armlink command option	No attribute	OVERLAY attribute	PROTECTED attribute
--merge	Merges all const strings.	Prevents merging across regions marked OVERLAY with other regions. const strings within a region are merged.	Prevents merging across regions marked PROTECTED with other regions. const strings within a region are merged.
--no_merge	Disables the merging of all const strings.	Disables the merging of all const strings.	Disables the merging of all const strings.
--merge_litpools	Merges all const values.	Prevents merging across regions marked OVERLAY. A const in an OVERLAY can be merged into a region that is not marked with either OVERLAY or PROTECTED. const values within a region are merged.	Prevents merging across regions marked PROTECTED with other regions. const values within a region are merged.
--no_merge_litpools	Disables the merging of all const values.	Disables the merging of all const values.	Disables the merging of all const values.

Related information

--merge, --no_merge
--merge_litpools, --no_merge_litpools
Merging identical constants
Load region attributes
Execution region attributes

10.2 Support for position independent code

Position Independent Code (PIC) permits an executable to be loaded at an address that is different from the static link time address.

PIC is either required or useful for a number of cases, including:

- Address space randomization.
- Shared libraries.
- Loadable modules.
- Flash/ROM construction from independent components.

Properties of PIC

There are a number of ways of implementing PIC, each with its own set of trade-offs.

Relocation required

Relocation, sometimes called rebasing, is where position independence can only be achieved by applying alterations to the program identified by relocations. In most models, the relocations are applied to the read/write part of the program, by an external program such

as a program loader, and applied once at load time. However, it is possible to bundle a loader into the program so that the program can relocate itself.

PI models requiring relocation by an external program are more flexible than those without, but they require you to build a more complex loader.

Online or offline position independence

The majority of PI applications are relocated at run-time when the application loads. In many cases the ELF file and its data structures are used by the run-time loader. It is also possible to construct a product out of components such as a hypervisor and guest operating systems. When building a flash image, it can help to construct the image from components that can be relocated when building the image, even if the addresses are fixed at run-time.

Shared Library Support or not

Supporting shared libraries presents some extra complexity. The library has its own code and data separate from the program, and its address might not be known to the program at static link time.

Fixed offset between code and data

A common implementation strategy, particularly when there is a Memory Management Unit (MMU) available, is to place the data for a program at a fixed offset away from the code. This strategy permits access to the data PC-relative with no relocations. This strategy might not work for Cortex®-M processors, because each instantiation of the program requires the code and data to be copied into RAM.

Data accessed through an offset from a static base

An alternative implementation strategy is also supported, particularly when there is no MMU available. In this strategy, place all the data in a contiguous block of memory and reserve a register, R9, as the static base. All data is accessed through offsets from the static base. This strategy does not require any relationship between code and data address, so code can be in flash and data can be at any point in RAM. The limitation of this strategy is that every program and shared library has its own static base, so implementing shared libraries with their own static data is more complicated.

For more information, see the Procedure Call Standard for the Arm Architecture.

PI code options in Arm Compiler for Embedded FuSa 6

Arm® Compiler for Embedded FuSa 6 supports a number of Position Independent Code (PIC) options.

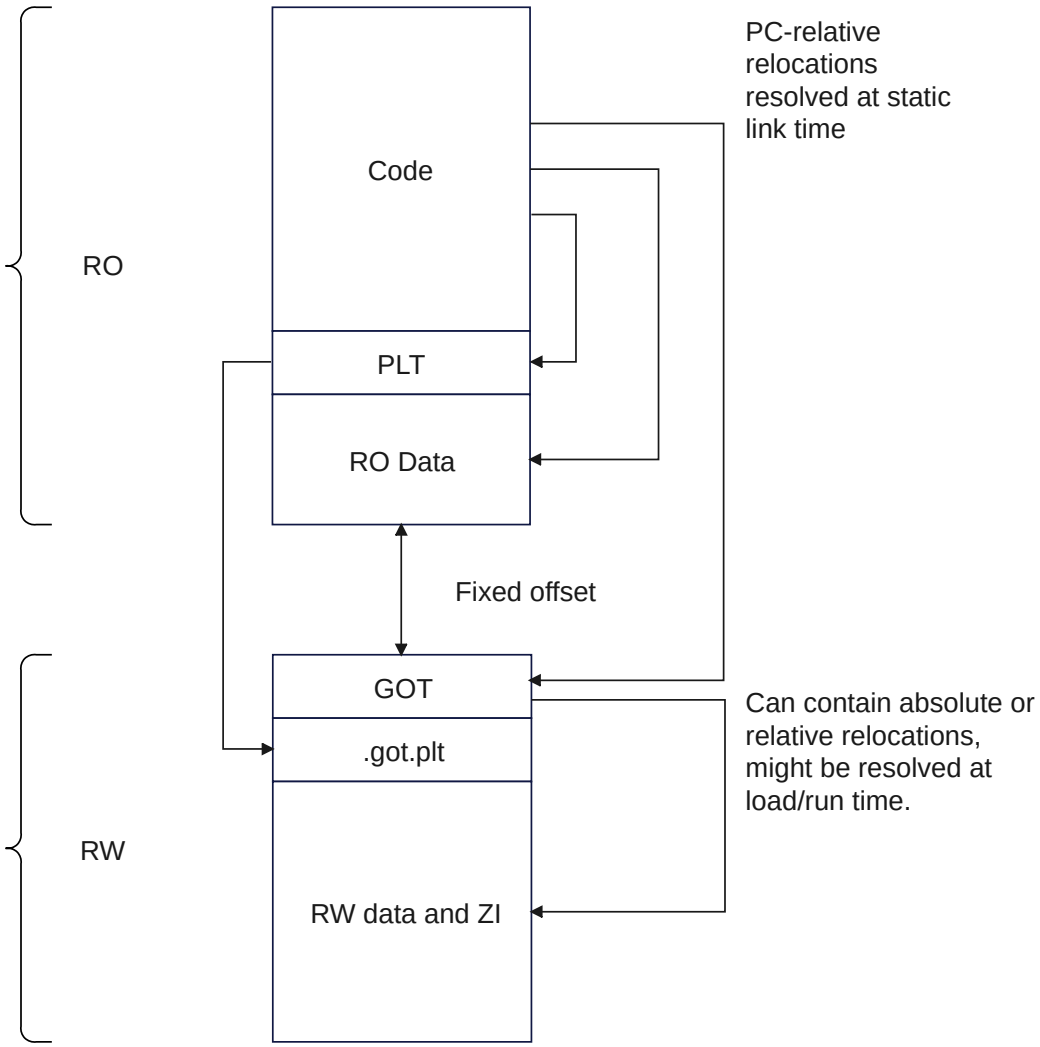
System V PIC and PIE

The PIC model is most often used on a platform OS where the ELF file is paged into memory and executed directly. The read-only (RO) part of the program is free from relocation, but the read/write (RW) part must be relocated. To achieve this distinction, the RO part only contains PC-relative offsets, and the RW part is a fixed distance away from the RO part. Therefore, the static linker can resolve the PC-relative offsets. Because the RO part of the program cannot use any absolute addresses, any time an absolute address is needed it must be redirected by way of the RW part. This redirection is performed by using a Global Offset Table (GOT) which is constructed at link time. Calling out to functions in other modules is achieved by a linker-

generated Procedure Linkage Table (PLT). Each PLT entry is a trampoline to load the address of the imported function from a RW part of the GOT sometimes called the `.got.plt`.

The following diagram shows a typical PIC memory layout:

Figure 10-1: Position Independent Code layout

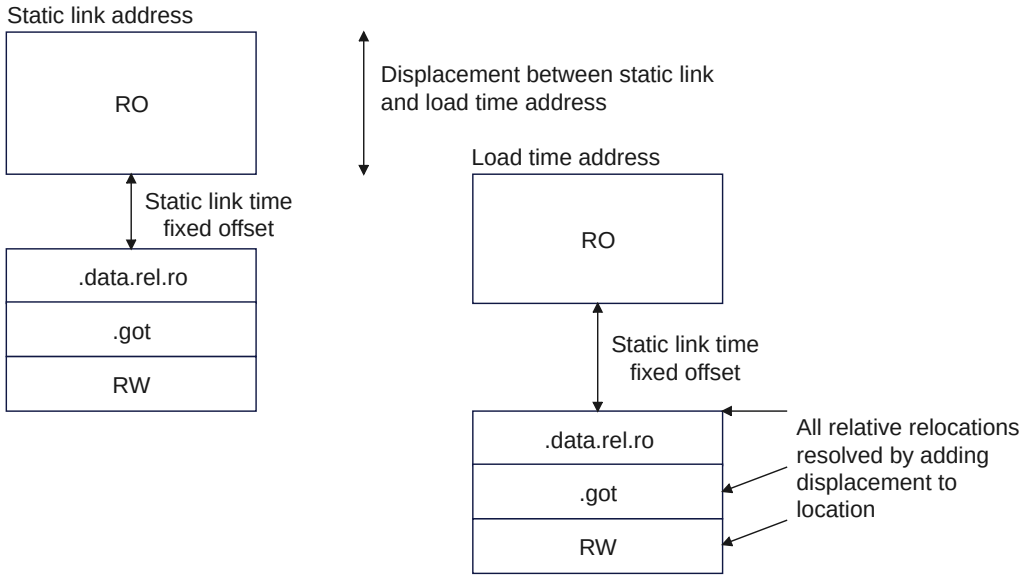


For a more thorough explanation of PIC, see Position Independent Code (PIC) in shared libraries. Although the examples are in X86_64, the general principle is the same.

When a dynamic relocation can be resolved without needing a symbol lookup, then the relocation can be expressed as `R_<ARCH>_RELATIVE`. For example, a relocation to a non-preemptable definition in the same module. To resolve an `R_<ARCH>_RELATIVE` relocation, a

loader only needs to add the displacement between the static link address and the address the program is being loaded at. This displacement is the same for all relative relocations.

Figure 10-2: Position Independent Code relative relocations



Use in bare-metal systems

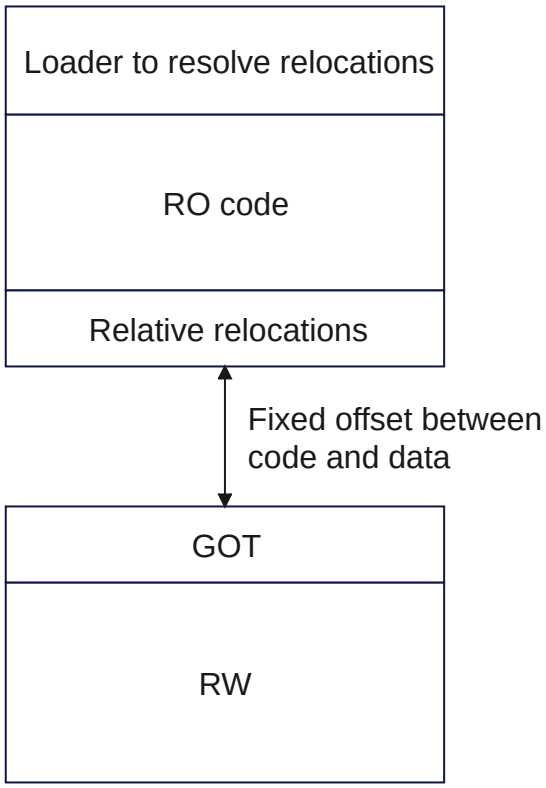
Code compiled with PIC must be linked into a suitable ELF file that maintains a fixed offset between code and data. `armlink` provides two ways to do this, `--sysv` and `--bare-metal-pie`:

- The `--sysv` option is intended for a sophisticated ELF loader that is able to resolve dynamic relocations. The details of writing such a loader are outside the scope of this document. For more information, see the section *Program Loading and Dynamic Linking* in the System V ABI for the Arm 64-bit Architecture (AArch64).
- The `--bare-metal-pie` option is limited to single position independent executables, but only needs a simple loader. See Bare-metal Position Independent Executables.

For systems without a MMU, the code and data must be copied into a contiguous free block of memory, maintaining the fixed offset from code to data. It is not possible to run code from flash and to have data in RAM.

A bare-metal Position Independent Executable (PIE) is an Arm Compiler for Embedded FuSa 6 only option that uses PIC addressing in the compiler. The linker constructs a self-relocating executable with the code a fixed offset from the data. This is essentially an implementation of static-pie in `armclang`.

Figure 10-3: Bare-metal PIE



Bare-metal PIE can support C++ because the relocations are fixed up by the loader. The main drawback is that the RO part and RW part have to be a fixed distance apart. This fixed separation can make it more difficult to deploy in single address space environments. The `armlink` option `--bare_metal_pie` is available to support the bare-metal PIE linking model.

Available `armclang` command-line options

- -fbare-metal-pie
- -fpic, -fno-pic
- -fsysv, -fno-sysv
- -shared

Available `armlink` command-line options

- --bare_metal_pie
- --bare_metal_sysv
- --fpic

- --shared
- --sysv

Read-Only Position Independent and Read/Write Position Independent

Read-Only Position Independent (ROPI) and Read/Write Position Independent (RWPI) code are separate options. Therefore, the following combinations are possible:

	no ROPI	ROPI
no RWPI	RO and RW data is accessed at an absolute address	RO data access is PC-relative RW data is accessed at an absolute address
RWPI	RO data is accessed at an absolute address RW data access is relative to a static base address	RO data access is PC-relative RW data access is relative to a static base address

In practice, the options are often used together because either all PI or no-PI is usually required.

The default configuration for ROPI and RWPI do not require relocations.

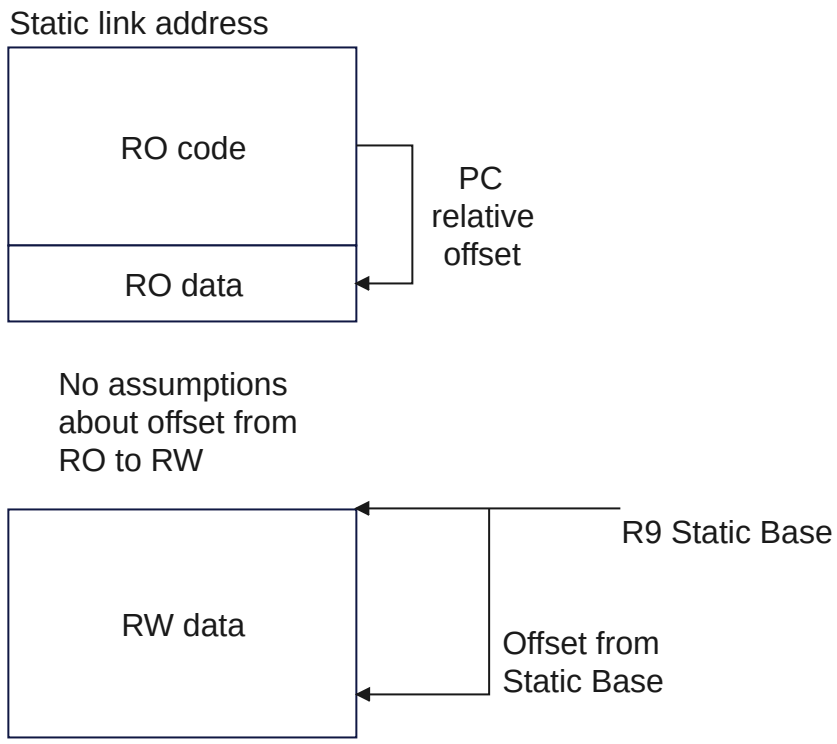
ROPI

Instead of loading the address of RO data, the compiler loads an offset from the PC to the RO data. This option means that the RO data must be placed at a fixed offset from the code at static link time.

RWPI

The platform register r9 becomes the static base register. This register points to the start of the static, RW, data for the program. All RW data are accessed using an offset from the static base register. This option means that the offset to any datum from the static base must be known at static link time.

Figure 10-4: ROPI and RWPI



Limitations of ROPI and RWPI

Static initialization involving addresses must be done at run-time because the static linker does not know the final addresses. RO data that needs a run-time initializer is emitted as RW.

Linking a program that has a ROPI and RWPI part and a non-ROPI and non-RWPI part is difficult. It is better to separate the ROPI and RWPI part and the non-ROPI and non-RWPI part into two programs.

C++ is not supported with ROPI and RWPI.

Not supported in AArch64 state.

Available `armclang` command-line options

- -fropi, -fno-ropi
- -frwpi, -fno-rwpi
- -fropi-lowering, -fno-ropi-lowering
- -frwpi-lowering, -fno-rwpi-lowering

Available `armlink` command-line options

- --piveneer, --no_piveneer
- --ropi
- --rwpi
- --ro_base
- --rw_base
- --rosplit
- --split

Position Independent eXecute Only

Position Independent eXecute Only (PIXO) is a generalization of RWPI that has a separate register for RO, called the RO Base. Therefore, separate RO and RW bases are available. This option permits the code to be execute-only. That is, the RO part is marked as readable and the RW part is marked as writeable. Apart from supporting execute-only, this option might not be useful to other use cases where sacrificing another register is less desirable.

Limitations of PIXO

The generation of PIXO libraries is only supported for Armv7-M targets.

Available `armclang` command-line options

- -mpixolib

Available `armlink` command-line options

- --pixolib

Related information

Bare-metal Position Independent Executables on page 200
SysV Dynamic Linking on page 396
Linking Models Supported by armlink
SysV Shared Libraries and Executables

10.3 Placing data items for target peripherals with a scatter file

To access the peripherals on your target, you must locate the data items that access them at the addresses of those peripherals.

About this task

To make sure that the data items are placed at the correct address for the peripherals, use the `__attribute__((section(".bss.ARM.__at_<address>")))` variable attribute together with a scatter file.

Procedure

1. Create `peripheral.c` to place the `my_peripheral` variable at address 0x10000000.

```
#include "stdio.h"

int my_peripheral __attribute__((section(".bss.ARM.__at_0x10000000"))) = 0;

int main(void)
{
    printf("%d\n",my_peripheral);
    return 0;
}
```

2. Create the scatter file `scatter.scat`.

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO 0x040000     ; load address = execution address
    {
        *(+RO +RW)     ; all RO sections (must include section with
                       ; initial entry point)
    }
    ; rest of scatter-loading description

    ARM_LIB_STACK 0x40000 EMPTY -0x20000  ; Stack region growing down
    { }
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
}

LR_2 0x01000000
{
    ER_ZI +0 UNINIT
    {
        *(.bss)
    }
}

LR_3 0x10000000
{
    ER_PERIPHERAL 0x10000000 UNINIT
    {
        *(.bss.ARM.__at_0x10000000)
    }
}
```

3. Build the image.

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 peripheral.c -g -c -o
 peripheral.o
armlink --cpu=cortex-m3 --scatter=scatter.scat --map --symbols peripheral.o --
output=peripheral.axf > map.txt
```

The memory map for load region `LR_3` is:

```
...
   Load Region LR_3 (Base: 0x10000000, Size: 0x00000000, Max: 0xffffffff,
 ABSOLUTE)

     Execution Region ER_PERIPHERAL (Base: 0x10000000, Size: 0x00000004, Max:
 0xffffffff, ABSOLUTE, UNINIT)

     Base Addr    Size         Type   Attr      Idx    E Section Name
 Object
```

```
    0x10000000   0x00000004   Zero   RW            6    .bss.ARM.__at_0x10000000
  peripheral.o
```

10.4 Placing the stack and heap with a scatter file

The Arm C library provides multiple implementations of the function `__user_setup_stackheap()`, and can select the correct one for you automatically from information that is given in a scatter file.

About this task

If you reimplement `__user_setup_stackheap()`, your version does not get invoked when stack and heap are defined in a scatter file.

You might have to update your startup code to use the correct initial stack pointer. Some processors, such as the Cortex®-M3 processor, require that you place the initial stack pointer in the vector table. See *Stack and heap configuration* in AN179 - Cortex-M3 Embedded Software Development for more details.

You must ensure correct alignment of the stack and heap:

- In AArch32 state, the stack and heap must be 8-byte aligned.
- In AArch64 state, the stack and heap must be 16-byte aligned.

Procedure

1. Define two special execution regions in your scatter file that are named `ARM_LIB_HEAP` and `ARM_LIB_STACK`.
2. Assign the `EMPTY` attribute to both regions.
   Because the stack and heap are in separate regions, the library selects the non-default implementation of `__user_setup_stackheap()` that uses the value of the symbols:
   - `Image$$ARM_LIB_STACK$$ZI$$Base`.
   - `Image$$ARM_LIB_STACK$$ZI$$Limit`.
   - `Image$$ARM_LIB_HEAP$$ZI$$Base`.
   - `Image$$ARM_LIB_HEAP$$ZI$$Limit`.

   You can specify only one `ARM_LIB_STACK` or `ARM_LIB_HEAP` region, and you must allocate a size.

```
LOAD_FLASH ...
{
    ...
    ARM_LIB_STACK 0x40000 EMPTY -0x20000  ; Stack region growing down
    { }
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
    ...
}
```

3. Alternatively, define a single execution region that is named `ARM_LIB_STACKHEAP` to use a combined stack and heap region. Assign the `EMPTY` attribute to the region.

Because the stack and heap are in the same region, `__user_setup_stackheap()` uses the value of the symbols `Image$$ARM_LIB_STACKHEAP$$ZI$$Base` and `Image$$ARM_LIB_STACKHEAP$$ZI$$Limit`.

10.5 Root region

A root region is a region with the same load and execution address. The initial entry point of an image must be in a root region.

If the initial entry point is not in a root region, the link fails and the linker gives an error message.


Note

All eXecute In Place (XIP) code must be stored in root regions.

Example

Root region with the same load and execution address.

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO 0x040000     ; load address = execution address
    {
        * (+RO)        ; all RO sections (must include section with
        ; initial entry point)
    }
    ...                ; rest of scatter-loading description
}
```

10.5.1 Effect of the ABSOLUTE attribute on a root region

You can use the `ABSOLUTE` attribute to specify a root region. This attribute is the default for an execution region.

To specify a root region, use `ABSOLUTE` as the attribute for the execution region. You can either specify the attribute explicitly or permit it to default, and use the same address for the first execution region and the enclosing load region.

To make the execution region address the same as the load region address, either:

- Specify the same numeric value for both the base address for the execution region and the base address for the load region.
- Specify a `+0` offset for the first execution region in the load region.

  If you specify an offset of zero (`+0`) for all subsequent execution regions in the load region, then all execution regions not following an execution region containing ZI are also root regions.

Example

The following example shows an implicitly defined root region:

```
LR_1 0x040000                   ; load region starts at 0x40000
{                               ; start of execution region descriptions
    ER_RO 0x040000 ABSOLUTE     ; load address = execution address
    {
        * (+RO)                 ; all RO sections (must include the section
                                ; containing the initial entry point)
    }
    ...                         ; rest of scatter-loading description
}
```

10.5.2 Effect of the FIXED attribute on a root region

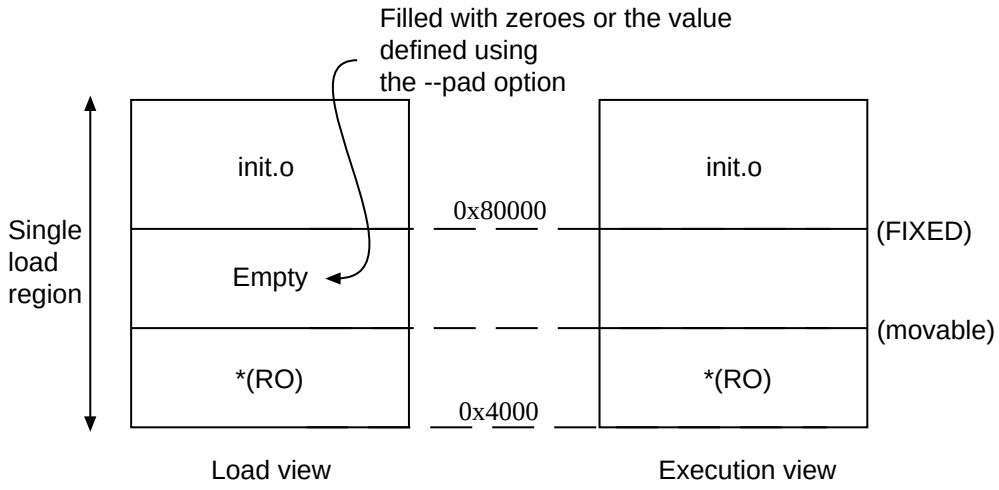
You can use the `FIXED` attribute for an execution region in a scatter file to create root regions that load and execute at fixed addresses.

Use the `FIXED` execution region attribute to ensure that the load address and execution address of a specific region are the same.

You can use the `FIXED` attribute to place any execution region at a specific address in ROM.

For example, the following memory map shows fixed execution regions:

Figure 10-5: Memory map for fixed execution regions



The following example shows the corresponding scatter-loading description:

```
LR_1 0x040000               ; load region starts at 0x40000
{                           ; start of execution region descriptions
```

```
    ER_RO 0x040000          ; load address = execution address
    {
        * (+RO)             ; RO sections other than those in init.o
    }
    ER_INIT 0x080000 FIXED ; load address and execution address of this
                           ; execution region are fixed at 0x80000
    {
        init.o(+RO)         ; all RO sections from init.o
    }
    ...                     ; rest of scatter-loading description
}
```

You can use this attribute to place a function or a block of data, for example a constant table or a checksum, at a fixed address in ROM. This makes it easier to access the function or block of data through pointers.

If you place two separate blocks of code or data at the start and end of ROM, some of the memory contents might be unused. For example, you might place some initialization code at the start of ROM and a checksum at the end of ROM. Use the `*` or `.ANY` module selector to flood fill the region between the end of the initialization block and the start of the data block.

To make your code easier to maintain and debug, use the minimum number of placement specifications in scatter files. Leave the detailed placement of functions and data to the linker.


Note

There are some situations where using `FIXED` and a single load region are not appropriate. Other techniques for specifying fixed locations are:

- If your loader can handle multiple load regions, place the RO code or data in its own load region.
- If you do not require the function or data to be at a fixed location in ROM, use `ABSOLUTE` instead of `FIXED`. The loader then copies the data from the load region to the specified address in RAM. `ABSOLUTE` is the default attribute.
- To place a data structure at the location of memory-mapped I/O, use two load regions and specify `UNINIT`. `UNINIT` ensures that the memory locations are not initialized to zero.

Example showing the misuse of the FIXED attribute

The following example shows common cases where the `FIXED` execution region attribute is misused:

```
LR1 0x8000
{
    ER_LOW +0 0x1000
    {
        *(+RO)
    }
; At this point the next available Load and Execution address is 0x8000 + size of
; contents of ER_LOW. The maximum size is limited to 0x1000 so the next available
 Load
; and Execution address is at most 0x9000
    ER_HIGH 0xF0000000 FIXED
    {
        *(+RW,+ZI)
    }
```

```
; The required execution address and load address is 0xF0000000. The linker inserts
; 0xF0000000 - (0x8000 + size of(ER_LOW)) bytes of padding so that load address
 matches
; execution address
}
; The other common misuse of FIXED is to give a lower execution address than the
 next
; available load address.
LR_HIGH 0x100000000
{
    ER_LOW 0x1000 FIXED
    {
        *(+RO)
    }
; The next available load address in LR_HIGH is 0x10000000. The required Execution
; address is 0x1000. Because the next available load address in LR_HIGH must
 increase
; monotonically the linker cannot give ER_LOW a Load Address lower than 0x10000000
}
```

10.6 Placing functions and data in a named section

You can place functions and data by separating them into their own objects without having to use toolchain-specific pragmas or attributes. Alternatively, you can specify a name of a section using the function or variable attribute, `__attribute__((section("<name>")))`.

About this task

You can use `__attribute__((section("<name>")))` to place a function or variable in a separate ELF section, where `<name>` is a name of your choice. You can then use a scatter file to place the named sections at specific locations.

You can place ZI data in a named section with `__attribute__((section(".bss.<name>")))`.

Use the following procedure to modify your source code to place functions and data in a specific section using a scatter file.

Procedure

1. Create a C source file `file.c` to specify a section name `foo` for a variable and a section name `.bss.mybss` for a zero-initialized variable `z`, for example:

```
#include "stdio.h"

int variable __attribute__((section("foo"))) = 10;
__attribute__((section(".bss.mybss"))) int z;

int main(void)
{
    int x = 4;
    int y = 7;
    z = x + y;
    printf("%d\n",variable);
    printf("%d\n",z);
    return 0;
}
```

2. Create a scatter file to place the named section, `scatter.scat`, for example:

```
LR_1 0x0
```

```
{
    ER_RO 0x0 0x4000
    {
        *(+RO)
    }
    ER_RW 0x4000 0x2000
    {
        *(+RW)
    }
    ER_ZI 0x6000 0x2000
    {
        *(+ZI)
    }
    ER_MYBSS 0x8000 0x2000
    {
        *(.bss.mybss)
    }

    ARM_LIB_STACK 0x40000 EMPTY -0x20000  ; Stack region growing down
    { }
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
}

FLASH 0x24000000 0x4000000
{
    ; rest of code

    ADDER 0x08000000
    {
        file.o (foo)                  ; select section foo from file.o
    }

}
```

The `ARM_LIB_STACK` and `ARM_LIB_HEAP` regions are required because the program is being linked with the semihosting libraries.

Note

If you omit `file.o (foo)` from the scatter file, the linker places the section in the region of the same type. That is, `ER_RW` in this example.

3. Compile and link the C source:

```
armclang --target=arm-arm-none-eabi -march=armv8-a file.c -g -c -O1 -o file.o
armlink --cpu=8-A.32 --scatter=scatter.scat --map file.o --output=file.axf
```

The `--map` option displays the memory map of the image.

In this example:

- `__attribute__((section("foo")))` specifies that the linker is to place the global variable `variable` in a section called `foo`.
- `__attribute__((section(".bss.mybss")))` specifies that the linker is to place the global variable `z` in a section called `.bss.mybss`.
- The scatter file specifies that the linker is to place the section `foo` in the `ADDER` execution region of the `FLASH` execution region.

The following example shows the output from `--map`:

```
...
    Execution Region ER_MYBSS (Base: 0x00008000, Size: 0x00000004, Max:
 0x00002000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name
 Object

    0x00008000   0x00000004   Zero   RW            7    .bss.mybss
 file.o
...
  Load Region FLASH (Base: 0x24000000, Size: 0x00000004, Max: 0x04000000,
 ABSOLUTE)

    Execution Region ADDER (Base: 0x08000000, Size: 0x00000004, Max: 0xffffffff,
 ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name
 Object

    0x08000000   0x00000004   Data   RW            5    foo
 file.o
...
```


Note

- If scatter-loading is not used, the linker places the section `foo` in the default `ER_RW` execution region of the `LR_1` load region. It also places the section `.bss.mybss` in the default execution region `ER_ZI`.
- If you have a scatter file that does not include the `foo` selector, then the linker places the section in the defined RW execution region.

You can also place a function at a specific address using `.ARM.__at_<address>` as the section name. For example, to place the function `sqr` at `0x20000`, specify:

```
int sqr(int n1) __attribute__((section(".ARM.__at_0x20000")));

int sqr(int n1)
{
    return n1*n1;
}
```

For more information, see Placement of functions and data at specific addresses.

Related information

Semihosting for AArch32 and AArch64

10.7 Loading armlink-generated ELF files that have complex scatter-files

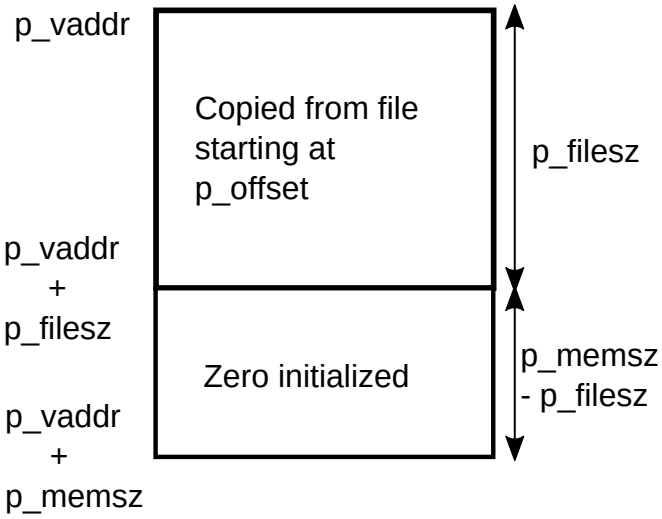
The information in program headers of type PT_LOAD is not always sufficient to load ELF files produced by armlink.

In the ELF specification, a PT_LOAD program header can be loaded by examining the fields:

- `p_offset`
- `p_vaddr`
- `p_paddr`. The value of this field is always the same as `p_vaddr` for armlink.
- `p_filesz`
- `p_memsz`

The ELF loader copies `p_filesz` bytes from the file at offset `p_offset` to the address specified by `p_vaddr`. The loader then creates `p_memsz - p_filesz` bytes of zero-initialized (ZI) data at address `p_vaddr + p_filesz`.

The final result is:



The scatter-loading notation permits ZI data to be created at a virtual address that is not at `p_vaddr + p_filesz`. Therefore, an ELF loader that creates ZI data by examining the fields of the program header alone creates the ZI data in the wrong place. To avoid this issue, do one of the following:

- Do not use the program headers to derive the execution view when loading the image onto the target device. Instead, use the fromelf utility to generate a binary file for the image, then load that binary file. The binary file contains a table containing the correct location of execution regions. The Arm C library uses this table to create the ZI data before program startup.
- Ensure that all execution regions are root regions with all the execution regions containing ZI data at end of the load region. You can check this situation by manually inspecting:
  - The output from `armlink --map`.

- The section headers in the output from `fromelf -v`.

The following example shows the behavior:

1. Create the file `foo.c` containing the following code:

```
int foo[0x10000];

int main(void)
{
  return foo[0];
}
```

2. Create the file `scatter.scat` containing the following load and execution regions:

```
LR 0x8000
{
    CODE +0
    {
        *(+RO)
    }
    RW_DATA +0
    {
        *(+RW)
    }
    /* ZI_DATA is not a root region */
    ZI_DATA 0x10000000
    {
        *(+ZI)
    }
}
LR_STACKHEAP 0x20000000
{
    ARM_LIB_STACKHEAP +0 EMPTY 0x2000 {}
}
```

3. Compile and link the example using the following commands:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -c foo.c -o foo.o

armlink --scatter=scatter.scat foo.o -o foo.axf
```

4. To examine the program headers, enter the following `fromelf` command:

```
fromelf -s -v foo.axf

...
============================================================================

** Program header #0

    Type          : PT_LOAD (1)
    File Offset   : 52 (0x34)
    Virtual Addr  : 0x00008000
    Physical Addr : 0x00008000
    Size in file  : 720 bytes (0x2d0)
    Size in memory: 262864 bytes (0x402d0)
    Flags         : PF_X + PF_W + PF_R + PF_ARM_ENTRY (0x80000007)
    Alignment     : 4
...
==================================

** Section #5
```

```
...
    179  foo                        0x10000000   Gb    2  Data  Hi   0x40000
...
```

In the output, `Program Header #0` describes the load region `LR`:

- `p_vaddr` field is the `Virtual Addr`
- `p_filesz` is the `Size in file`
- `p_memsz` is the `Size in memory`.

If you use an ELF loader to create the memory based on the program header, then `0x402d0 - 0x2d0` bytes of ZI data are created at address `0x8000 + 0x2d0`. This address does not match the expected execution address of `0x10000000` as shown by the address of symbol `foo`.

10.8 Placement of functions and data at specific addresses

You can place a single function or data item at a fixed address. You must enable the linker to process the function or data separately from the other input files.

Where they are required, the compiler normally produces RO, RW, and ZI sections from a single source file. These sections contain all the code and data from the source file.


Note

For images targeted at Arm®v6-M, Armv7-M, or Armv8-M, the compiler might generate eXecute-Only (XO) sections.

Typically, you create a scatter file that defines an execution region at the required address with a section description that selects only one section.

To place a function or variable at a specific address, it must be placed in its own section. There are several ways to place a function or variable in its own section:

- By default, the compiler places each function and variable in individual ELF sections. To override this default placement, use the `-fno-function-sections` or `-fno-data-sections` compiler options.
- Place the function or data item in its own source file.
- Use `__attribute__((section("<name>")))` to place functions and variables in a specially named section, `.ARM.__at_<address>`, where `<address>` is the address to place the function or variable. For example, `__attribute__((section(".ARM.__at_0x4000")))`.

  To place ZI data at a specific address, use the variable attribute `__attribute__((section("<name>")))` with the special name `.bss.ARM.__at_<address>`.

  These specially named sections are called `__at` sections.
- Use the `.section` directive from assembly language. In assembly code, the smallest locatable unit is a `.section`.

10.8.1 Placement of __at sections at a specific address

You can give a section a special name that encodes the address where it must be placed.

To place a section at a specific address, use the function or variable attribute `__attribute__((section("<name>")))` with the special name `.ARM.__at_<address>`.

To place ZI data at a specific address, use the variable attribute `__attribute__((section("<name>")))` with the special name `.bss.ARM.__at_<address>`.

`<address>` is the required address of the section. The compiler normalizes this address to eight hexadecimal digits. You can specify the address in hexadecimal or decimal. Sections in the form of `.ARM.__at_<address>` are referred to by the abbreviation `__at`.

The following example shows how to assign a variable to a specific address in C or C++ code:

```
// place variable1 in a section called .ARM.__at_0x8000
int variable1 __attribute__((section(".ARM.__at_0x8000"))) = 10;
```


Note

The name of the section is only significant if you are trying to match the section by name in a scatter file. Without overlays, the linker automatically assigns `__at` sections when you use the `--autoat` command-line option. This option is the default. If you are using overlays, then you cannot use `--autoat` to place `__at` sections.

Supporting arithmetic expressions for an address when placing __at sections

If you need to use an arithmetic expression to specify the section address, then you cannot use the `__attribute__((section(".ARM.__at_<address>")))` attribute. Instead, you must use a pointer approach.

For example, to specify the address as `0xE0001000 + MY_PREDEFINED_OFFSET`, then use the following code:

```
static my_variable_type * const my_address = (my_variable_type *) (0xE0001000 +
 MY_PREDEFINED_OFFSET);

#define my_variable (*my_address)
```

Related information

Placement of functions and data at specific addresses on page 190
Restrictions on placing __at sections on page 191

10.8.2 Restrictions on placing __at sections

There are restrictions when placing `__at` sections at specific addresses.

The following restrictions apply:

- `__at` section address ranges must not overlap, unless the overlapping sections are placed in different overlay regions.
- `__at` sections are not permitted in position independent execution regions.
- You must not reference the linker-defined symbols `$$Base`, `$$Limit` and `$$Length` of an `__at` section.
- `__at` sections must have an address that is a multiple of their alignment.
- `__at` sections ignore any `+FIRST` or `+LAST` ordering constraints.

10.8.3 Automatic placement of __at sections

The automatic placement of `__at` sections is enabled by default. Use the linker command-line option, `--no_autoat` to disable this feature.

Note

You cannot use `__at` section placement with position independent execution regions.

When linking with the `--autoat` option, the linker does not place `__at` sections with scatter-loading selectors. Instead, the linker places the `__at` section in a compatible region. If no compatible region is found, the linker creates a load region and an execution region for the `__at` section.

All linker execution regions created by `--autoat` have the `UNINIT` scatter-loading attribute. If you require a Zero-Initialized (ZI) `__at` section to be zero-initialized, then it must be placed within a compatible region. A linker execution region created by `--autoat` must have a base address that is at least 4 byte-aligned. If any region is incorrectly aligned, the linker produces an error message.

A compatible region is one where:

- The `__at` address lies within the execution region base and limit, where limit is the base address + maximum size of execution region. If no maximum size is set, the linker sets the limit for placing `__at` sections as the current size of the execution region without `__at` sections plus a constant. The default value of this constant is `10240` bytes, but you can change the value using the `--max_er_extension` command-line option.
- The execution region meets at least one of the following conditions:
  - It has a selector that matches the `__at` section by the standard scatter-loading rules.
  - It has at least one section of the same type (RO or RW) as the `__at` section.
  - It does not have the `EMPTY` attribute.

Note

The linker considers an __at section with type RW compatible with RO.

The following example shows the sections .ARM.__at_0x0000 type RO, .ARM.__at_0x4000 type RW, and .ARM.__at_0x8000 type RW:

```
// place the RO variable in a section called .ARM.__at_0x0000
const int foo __attribute__((section(".ARM.__at_0x0000"))) = 10;

// place the RW variable in a section called .ARM.__at_0x4000
int bar __attribute__((section(".ARM.__at_0x4000"))) = 100;

// place "variable" in a section called .ARM.__at_0x00008000
int variable __attribute__((section(".ARM.__at_0x00008000")));
```

The following scatter file shows how automatically to place these __at sections:

```
LR1 0x0
{
    ER_RO 0x0 0x4000
    {
        *(+RO)      ; .ARM.__at_0x0000 lies within the bounds of ER_RO
    }
    ER_RW 0x4000 0x2000
    {
        *(+RW)      ; .ARM.__at_0x4000 lies within the bounds of ER_RW
    }
    ER_ZI 0x6000 0x2000
    {
        *(+ZI)
    }
}
; The linker creates a load region and an execution region for the __at section
; .ARM.__at_0x8000 because it lies outside all candidate regions.
```

10.8.4 Manual placement of __at sections

You can have direct control over the placement of __at sections, if required.

You can use the standard section-placement rules to place __at sections when using the --no_autoat command-line option.

Note

You cannot use __at section placement with position independent execution regions.

The following example shows the placement of read-only sections `.ARM.__at_0x2000` and the read-write section `.ARM.__at_0x4000`. Load and execution regions are not created automatically in manual mode. An error is produced if an `__at` section cannot be placed in an execution region.

The following example shows the placement of the variables in C or C++ code:

```
// place the RO variable in a section called .ARM.__at_0x2000
const int foo __attribute__((section(".ARM.__at_0x2000"))) = 100;
// place the RW variable in a section called .ARM.__at_0x4000
int bar __attribute__((section(".ARM.__at_0x4000")));
```

The following scatter file shows how to place `__at` sections manually:

```
LR1 0x0
{
    ER_RO 0x0 0x2000
    {
        *(+RO)               ; .ARM.__at_0x0000 is selected by +RO
    }
    ER_RO2 0x2000
    {
        *(.ARM.__at_0x02000)  ; .ARM.__at_0x2000 is selected by the section named
                              ; .ARM.__at_0x2000
    }
    ER2 0x4000
    {
        *(+RW, +ZI)           ; .ARM.__at_0x4000 is selected by +RW
    }
}
```

10.8.5 Place a key in flash memory with an __at section

Some flash devices require a key to be written to an address to activate certain features. An `__at` section provides a simple method of writing a value to a specific address.

Placing the flash key variable in C or C++ code

Assume that a device has flash memory from `0x8000` to `0x10000` and a key is required in address `0x8000`. To do this with an `__at` section, you must declare a variable so that the compiler can generate a section called `.ARM.__at_0x8000`.

```
// place flash_key in a section called .ARM.__at_0x8000
long flash_key __attribute__((section(".ARM.__at_0x8000")));
```

Manually placing a flash execution region

The following example shows how to manually place a flash execution region with a scatter file:

```
ER_FLASH 0x8000 0x2000
{
    *(+RW)
    *(.ARM.__at_0x8000) ; key
}
```

Use the linker command-line option `--no_autoat` to enable manual placement.

Automatically placing a flash execution region

The following example shows how to automatically place a flash execution region with a scatter file. Use the linker command-line option `--autoat` to enable automatic placement.

```
LR1 0x0
{
    ER_FLASH 0x8000 0x2000
    {
        *(+RO)                  ; other code and read-only data, the
                                ; __at section is automatically selected
    }
    ER2 0x4000
    {
        *(+RW +ZI)              ; Any other RW and ZI variables
    }
}
```

10.8.6 Placing constants at fixed locations

There are some situations when you want to place constants at fixed memory locations. For example, you might want to write a value to FLASH to read-protect a SoC device.

Procedure

1. Create a C file `abs_address.c` to define an integer and a string constant.

```
unsigned int const number = 0x12345678;
char* const string = "Hello World";
```

2. Create a scatter file, `scatter.scat`, to place the constants in separate sections `ER_RONUMBERS` and `ER_ROSTRINGS`.

```
LR_1 0x040000          ; load region starts at 0x40000
{                      ; start of execution region descriptions
    ER_RO 0x040000     ; load address = execution address
    {
        *(+RO +RW)     ; all RO sections (must include section with
                       ; initial entry point)
    }
    ER_RONUMBERS +0
    {
        *(.rodata.number, +RO-DATA)
    }
    ER_ROSTRINGS +0
    {
        *(.rodata.string, .rodata.str1.1, +RO-DATA)
    }
                       ; rest of scatter-loading description

    ARM_LIB_STACK 0x80000 EMPTY -0x20000  ; Stack region growing down
    { }
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
}
```

`armclang` puts string literals in a section called `.rodata.str1.1`

3. Compile and link the file.

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a9 abs_address.c -g -c -o
 abs_address.o
armlink --cpu=cortex-a9 --scatter=scatter.scat abs_address.o --
output=abs_address.axf
```

4. Run `fromelf` on the image to view the contents of the output sections.

```
fromelf -c -d abs_address.axf
```

   The output contains the following sections:

```
...

** Section #2 'ER_RONUMBERS' (SHT_PROGBITS) [SHF_ALLOC]
    Size   : 4 bytes (alignment 4)
    Address: 0x00040000

    0x040000:   78 56 34 12                                      xV4.

** Section #3 'ER_ROSTRINGS' (SHT_PROGBITS) [SHF_ALLOC]
    Size   : 16 bytes (alignment 4)
    Address: 0x00040004

    0x040004:   48 65 6c 6c 6f 20 57 6f 72 6c 64 00 04 00 04 00    Hello
 World.....
...
```

5. Replace the `ER_RONUMBERS` and `ER_ROSTRINGS` sections in the scatter file with the following `ER_RODATA` section:

```
ER_RODATA +0
{
    abs_address.o(.rodata.number, .rodata.string, .rodata.str1.1, +RO-DATA)
}
```

6. Repeat steps 3 and 4.
   The integer and string constants are both placed in the `ER_RODATA` section, for example:

```
** Section #2 'ER_RODATA' (SHT_PROGBITS) [SHF_ALLOC]

    Size   : 20 bytes (alignment 4)
    Address: 0x00040000

    0x040000:   78 56 34 12 48 65 6c 6c 6f 20 57 6f 72 6c 64 00    xV4.Hello
 World.
    0x040010:   04 00 04 00                                        ....
```

10.8.7 Placing jump tables in ROM

You might find that jump tables are placed in RAM rather than in ROM.

About this task

A jump table might be placed in a RAM `.data` section when you define it as follows:

```
typedef void PFUNC(void);
const PFUNC *table[3] = {func0, func1, func2};
```

The compiler also issues the warning:

```
jump.c:19:1: warning: 'const' qualifier on function type 'PFUNC'

      (aka 'void (void)') has unspecified behavior
const PFUNC *table[3] = {func0, func1, func2};
^~~~~~
```

The following procedure describes how to place the jump table in a ROM `.rodata` section.

Procedure

1. Create a C file jump.c .
   Make the `PFUNC` type a pointer to a void function that has no parameters. You can then use `PFUNC` to create an array of constant function pointers.

   ```
   extern void func0(void);
   extern void func1(void);
   extern void func2(void);

   typedef void (*PFUNC)(void);

   const PFUNC table[] = {func0, func1, func2};

   void jump(unsigned i)
   {
     if (i<=2)
     table[i]();
   }
   ```

2. Compile the file.

   ```
   armclang --target=arm-arm-none-eabi -mcpu=cortex-a9 jump.c -g -c -o jump.o
   ```

3. Run `fromelf` on the image to view the contents of the output sections.

   ```
   fromelf -c -d jump.o
   ```

   The table is placed in the read-only section `.rodata` that you can place in ROM as required:

   ```
   ...

   ** Section #3 '.text.jump' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
       Size   : 64 bytes (alignment 4)
       Address: 0x00000000

       $a.0
       [Anonymous symbol #24]
       jump
           0x00000000:    e92d4800    .H-.    PUSH     {r11,lr}
           0x00000004:    e24dd008    ..M.    SUB      sp,sp,#8
           0x00000008:    e1a01000    ....    MOV      r1,r0
           0x0000000c:    e58d0004    ....    STR      r0,[sp,#4]
           0x00000010:    e3500002    ..P.    CMP      r0,#2
           0x00000014:    e58d1000    ....    STR      r1,[sp,#0]
           0x00000018:    8a000006    ....    BHI      {pc}+0x20 ; 0x38
           0x0000001c:    eaffffff    ....    B        {pc}+0x4 ; 0x20
           0x00000020:    e59d0004    ....    LDR      r0,[sp,#4]
           0x00000024:    e3001000    ....    MOVW     r1,#:LOWER16: table
           0x00000028:    e3401000    ..@.    MOVT     r1,#:UPPER16: table
           0x0000002c:    e7910100    ....    LDR      r0,[r1,r0,LSL #2]
           0x00000030:    e12fff30    0./.    BLX      r0
   ```

```
        0x00000034:    eafffffc    ....    B        {pc}+0x4 ; 0x38
        0x00000038:    e28dd008    ....    ADD      sp,sp,#8
        0x0000003c:    e8bd8800    ....    POP      {r11,pc}

...
** Section #7 '.rodata.table' (SHT_PROGBITS) [SHF_ALLOC]
    Size   : 12 bytes (alignment 4)
    Address: 0x00000000

    0x000000:   00 00 00 00 00 00 00 00 00 00 00 00                ............
...
```

10.8.8 Placing a variable at a specific address without scatter-loading

This example shows how to modify your source code to place code and data at specific addresses, and does not require a scatter file.

To place code and data at specific addresses without a scatter file:

1. Create the source file `main.c` containing the following code:

```
#include <stdio.h>

extern int sqr(int n1);
const int gValue __attribute__((section(".ARM.__at_0x5000"))) = 3; // Place at
 0x5000
int main(void)
{
    int squared;
    squared=sqr(gValue);
    printf("Value squared is: %d\n", squared);
    return 0;
}
```

2. Create the source file `function.c` containing the following code:

```
int sqr(int n1)
{
    return n1*n1;
}
```

3. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c function.c
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --map function.o main.o -o squared.axf
```

   The `--map` option displays the memory map of the image. Also, `--autoat` is the default.

In this example, `__attribute__((section(".ARM.__AT_0x5000")))` specifies that the global variable `gValue` is to be placed at the absolute address `0x5000`. `gValue` is placed in the execution region `ER$$.ARM.__AT_0x5000` and load region `LR$$.ARM.__AT_0x5000`.

The memory map shows:

```
...
```

```
 Load Region LR$$.ARM.__AT_0x5000 (Base: 0x00005000, Size: 0x00000004, Max:
0x00000004, ABSOLUTE)

   Execution Region ER$$.ARM.__AT_0x5000 (Base: 0x00005000, Size: 0x00000004, Max:
0x00000004, ABSOLUTE, UNINIT)

   Base Addr    Size         Type   Attr      Idx    E Section Name        Object

   0x00005000   0x00000004   Data   RO           18    .ARM.__AT_0x5000  main.o
```

10.8.9 Placing a variable at a specific address with scatter-loading

This example shows how to modify your source code to place code and data at a specific address using a scatter file.

To modify your source code to place code and data at a specific address using a scatter file:

1. Create the source file `main.c` containing the following code:

```
#include <stdio.h>
extern int sqr(int n1);
// Place at address 0x10000
const int gValue __attribute__((section(".ARM.__at_0x10000"))) = 3;
int main(void)
{
    int squared;
    squared=sqr(gValue);
    printf("Value squared is: %d\n", squared);
    return 0;
}
```

2. Create the source file `function.c` containing the following code:

```
int sqr(int n1)
{
    return n1*n1;
}
```

3. Create the scatter file `scatter.scat` containing the following load region:

```
LR1 0x0
{
    ER1 0x0
    {
        *(+RO)                      ; rest of code and read-only data
    }
    ER2 +0
    {
        function.o
        *(.ARM.__at_0x10000)        ; Place gValue at 0x10000
    }
    ; RW and ZI data to be placed at 0x200000
    RAM 0x200000 (0x1FF00-0x2000)
    {
        *(+RW, +ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
```

```
    }
}
```

The `ARM_LIB_STACK` and `ARM_LIB_HEAP` regions are required because the program is being linked with the semihosting libraries.

4. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c function.c
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --no_autoat --scatter=scatter.scat --map function.o main.o -o squared.axf
```

The `--map` option displays the memory map of the image.

The memory map shows that the variable is placed in the `ER2` execution region at address `0x10000`:

```
...
    Execution Region ER2 (Base: 0x00002a54, Size: 0x0000d5b0, Max: 0xffffffff,
ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00002a54   0x0000001c   Code   RO            4    .text.sqr
function.o
    0x00002a70   0x0000d590   PAD
    0x00010000   0x00000004   Data   RO            9    .ARM.__at_0x10000   main.o
```

In this example, the size of `ER1` is unknown. Therefore, `gValue` might be placed in `ER1` or `ER2`. To make sure that `gValue` is placed in `ER2`, you must include the corresponding selector in `ER2` and link with the `--no_autoat` command-line option. If you omit `--no_autoat`, `gValue` is placed in a separate load region `LR$$.ARM.__at_0x10000` that contains the execution region `ER$$.ARM.__at_0x10000`.

Related information

Semihosting for AArch32 and AArch64

10.9 Bare-metal Position Independent Executables

A bare-metal Position Independent Executable (PIE) is an executable that does not need to be executed at a specific address. It can be executed at any suitably aligned address.

Note

`armclang` supports the `-fropi` and `-frwpi` options. You can use these options to create bare-metal position independent executables.

Position independent code uses PC-relative addressing modes where possible and otherwise accesses global data via the Global Offset Table (GOT). The address entries in the GOT and initialized pointers in the data area are updated with the executable load address when the executable runs for the first time.

All objects and libraries that are linked into the image must be compiled to be position independent.

Compiling and linking a bare-metal PIE

Create `hello.c` containing the following code:

```
#include <stdio.h>

int main(void)
{
  printf("Hello World!\n");
  return 0;
}
```

To compile and automatically link this code for bare-metal PIE, use the `-fbare-metal-pie` option with `armclang`:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -fbare-metal-pie hello.c -o hello
```

Alternatively, you can compile with the `armclang` option `-fbare-metal-pie` and link with the `armlink` option `--bare_metal_pie` as separate steps:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -fbare-metal-pie -c hello.c
armlink --bare_metal_pie hello.o -o hello
```

The resulting executable `hello` is a bare-metal PIE.

Note

Legacy code that is compiled with `armcc` to be included in a bare-metal PIE must be compiled with either the option `--apcs=/fpic` or, if it contains no references to global data, the option `--apcs=/ropi`.

If you are using Link-Time Optimization (LTO), use the `armlink` option `--lto_relocation_model=pic` to tell the link time optimizer to produce position independent code:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -flto -fbare-metal-pie -c hello.c
 -o hello.bc
armlink --lto --lto_relocation_model=pic --bare_metal_pie hello.bc -o hello
```

Restrictions

A bare-metal PIE executable must conform to the following:

- The `.got` section must be placed in a writable region.
- All references to symbols must be resolved at link time.
- The image must be linked Position Independent with a base address of 0x0.
- The code and data must be linked at a fixed offset from each other.

- The stack must be set up before the runtime relocation routine `__arm_relocate_pie_` is called. This means that the stack initialization code must only use PC-relative addressing if it is part of the image code.
- It is the responsibility of the target platform that loads the PIE to ensure that the ZI region is zero-initialized.
- The scatter file load region attribute `PI` is not supported for AArch64 state.
- Mixing absolute linked and bare-metal PIE images is not supported. You must link them as two separate units.
- When writing assembly code for position independence, some instructions such as `LDR` let you specify a label for a PC-relative address. For example:

```
    ldr r0,=__main
```

  Specifying a label causes the link step to fail when building with `--bare-metal-pie`, because the symbol is in a read-only section. `armlink` returns an error message, for example:

```
Error: L6084E: Dynamic relocation from #REL:0 in unwritable section
 foo-7cb47a.o(.text.main) of type R_ARM_RELATIVE to symbol main cannot be
 applied.
```

  The workaround is to specify symbols indirectly in a writable section, for example:

```
    ldr r0, __main_addr
...
    .type __main_addr, %object
    .data
__main_addr:
    .word __main
    .end
```

Using a scatter file

An example scatter file is:

```
LR 0x0
{
    er_ro +0 {
        *(+RO)
    }
    DYNAMIC_RELOCATION_TABLE +0 {
        *(DYNAMIC_RELOCATION_TABLE)
    }
    got +0 {
        *(.got)
    }
    er_rw +0 {
        *(+RW)
    }
    er_zi +0 {
        *(+ZI)
    }

    ; Add any stack and heap section required by the user supplied
    ; stack/heap initialization routine here
}
```

Use the `armlink` option `--bare-metal-pie`, or use either the `--sysv` or `--shared` option with `--fpic`.


Note

For AArch32 state, you can include the `PI` attribute for the load region, for example `LR 0x0 PI`.

The linker generates the `DYNAMIC_RELOCATION_TABLE` section. This section must be placed in an execution region called `DYNAMIC_RELOCATION_TABLE`. This allows the runtime relocation routine `__arm_relocate_pie_` that is provided in the C library to locate the start and end of the table using the symbols `Image$$DYNAMIC_RELOCATION_TABLE$$Base` and `Image$$DYNAMIC_RELOCATION_TABLE$$Limit`.

When using a scatter file and the default entry code that the C library supplies, the linker requires that you provide your own routine for initializing the stack and heap. This user supplied stack and heap routine is run before the routine `__arm_relocate_pie_`. Therefore, it is necessary to ensure that this routine only uses PC relative addressing.

Related information

--fpic (armlink)
--pie (armlink)
--bare_metal_pie (armlink)
--bare_metal_sysv (armlink)
--ref_pre_init (armlink)
--sysv
-fbare-metal-pie (armclang)
-fropi (armclang)
-frwpi (armclang)
Load region attributes

10.10 Placement of Arm C and C++ library code

You can place code from the Arm standard C and C++ libraries using a scatter file.

Use `*armlib*` or `*libcxx*` so that the linker can resolve library naming in your scatter file.

Some Arm C and C++ library sections must be placed in a root region, for example `__main.o`, `__scatter*.o`, `__dc*.o`, and `*Region$$Table`. This list can change between releases. The linker can place all these sections automatically in a future-proof way with `InRoot$$Sections`.

Note

For AArch64, `__rtentry*.o` is moved to a root region.

Related information

Region table format

10.10.1 Placement of code in a root region

Some code must always be placed in a root region. You do this in a similar way to placing a named section.

To place all sections that must be in a root region, use the section selector `InRoot$$Sections`. For example :

```
ROM_LOAD 0x0000 0x4000
{
  ROM_EXEC 0x0000 0x4000       ; root region at 0x0
  {
    vectors.o (Vect, +FIRST)   ; Vector table
    * (InRoot$$Sections)       ; All library sections that must be in a
                               ; root region, for example, __main.o,
                               ; __scatter*.o, __dc*.o, and *Region$$Table
  }
  RAM 0x10000 0x8000
  {
    * (+RO, +RW, +ZI)          ; all other sections
  }
}
```

Related information

Region table format

10.10.2 Placement of Arm C library code

You can place C library code using a scatter file.

To place C library code, specify the library path and library name as the module selector. You can use wildcard characters if required. For example:

```
LR1 0x0
{
    ROM1 0
    {
        * (InRoot$$Sections)
        * (+RO)
    }
    ROM2 0x1000
    {
        *armlib/c_* (+RO)                    ; all Arm-supplied C library functions
    }
```

```
    RAM1 0x3000
    {
        *armlib* (+RO)                          ; all other Arm-supplied library code
                                                ; for example, floating-point libraries
    }
    RAM2 0x4000
    {
        * (+RW, +ZI)
    }
}
```

The name `armlib` indicates the Arm C library files that are located in the directory `<install_directory>\lib\armlib`.

10.10.3 Placing Arm C++ library code

You can place C++ library code using a scatter file.

About this task

To place C++ library code, specify the library path and library name as the module selector. You can use wildcard characters if required.

Procedure

1. Create the following C++ program, `foo.cpp`:

```
#include <iostream>

using namespace std;

extern "C" int foo ()
{
  cout << "Hello" << endl;
  return 1;
}
```

2. To place the C++ library code, define the following scatter file, `scatter.scat`:

```
LR 0x8000
{
    ER1 +0
    {
        *armlib*(+RO)
    }
    ER2 +0
    {
        *libcxx*(+RO)
    }
    ER3 +0
    {
        *(+RO)

        ; All .ARM.exidx* sections must be coalesced into a single contiguous
        ; .ARM.exidx section because the unwinder references linker-generated
        ; Base and Limit symbols for this section.
        *(0x70000001)  ; SHT_ARM_EXIDX sections

        ; All .init_array sections must be coalesced into a single contiguous
        ; .init_array section because the initialization code references
        ; linker-generated Base and Limit for this section.
        *(.init_array)
    }
    ER4 +0
```

```
    {
        *(+RW,+ZI)
    }
}
```

The name `*armlib*` matches `<install_directory>\lib\armlib`, indicating the Arm C library files that are located in the `armlib` directory.

The name `*libcxx*` matches `<install_directory>\lib\libcxx`, indicating the C++ library files that are located in the `libcxx` directory.

3. Compile and link the sources:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c foo.cpp
armclang --target=arm-arm-none-eabi -march=armv8-a -c main.c
armlink --scatter=scatter.scat --map main.o foo.o -o foo.axf
```

The `--map` option displays the memory map of the image.

10.11 Manual placement of unassigned sections

The linker attempts to place Input sections into specific execution regions. For any Input sections that cannot be resolved, and where the placement of those sections is not important, you can specify where the linker is to place them.

To place sections that are not automatically assigned to specific execution regions, use the `.ANY` module selector in a scatter file.

Usually, a single `.ANY` selector is equivalent to using the `*` module selector. However, unlike `*`, you can specify `.ANY` in multiple execution regions.

The linker has default rules for placing unassigned sections when you specify multiple `.ANY` selectors. You can override the default rules using the following command-line options:

- `--any_contingency` to permit extra space in any execution regions containing `.ANY` sections for linker-generated content such as veneers and alignment padding.
- `--any_placement` to provide more control over the placement of unassigned sections.
- `--any_sort_order` to control the sort order of unassigned Input sections.

Note

The placement of data can cause some data to be removed and shrink the size of the sections.

In a scatter file, you can also:

- Assign a priority to a `.ANY` selector to give you more control over how the unassigned sections are divided between multiple execution regions. You can assign the same priority to more than one execution region.

- Specify the maximum size for an execution region that the linker can fill with unassigned sections.

The following are relevant operations in the linking process and their order:

1. `.ANY` placement.
2. String merging.
3. Region table creation.
4. Late library load (scatter-load functions).
5. Veneer generation + literal pool merging.

String and literal pool merging can reduce execution size, while region table creation, late library load, and veneer generation can increase it. Padding also affects the execution size of the region.


Note

Extra, more-specific operations can also increase or decrease execution size after the `.ANY` placement, such as the generation of PLT/GOT and exception-section optimizations.

10.11.1 Default rules for placing unassigned sections

The linker has default rules for placing sections when using multiple `.ANY` selectors.

When more than one `.ANY` selector is present in a scatter file, the linker sorts sections in descending size order. It then takes the unassigned section with the largest size and assigns the section to the most specific `.ANY` execution region that has enough free space. For example, `.ANY(.text)` is judged to be more specific than `.ANY(+RO)`.

If several execution regions are equally specific, then the section is assigned to the execution region with the most available remaining space.

For example:

- You might have two equally specific execution regions where one has a size limit of `0x2000` and the other has no limit. In this case, all the sections are assigned to the second unbounded `.ANY` region.
- You might have two equally specific execution regions where one has a size limit of `0x2000` and the other has a size limit of `0x3000`. In this case, the first sections to be placed are assigned to the second `.ANY` region of size limit `0x3000`. This assignment continues until the remaining size of the second `.ANY` region is reduced to `0x2000`. From this point, sections are assigned alternately between both `.ANY` execution regions.

You can specify a maximum amount of space to use for unassigned sections with the execution region attribute `ANY_SIZE`.

10.11.2 Command-line options for controlling the placement of unassigned sections

You can modify how the linker places unassigned input sections when using multiple `.ANY` selectors by using a different placement algorithm or a different sort order.

The following command-line options are available:

- `--any_placement=<algorithm>`, where `<algorithm>` is one of `first_fit`, `worst_fit`, `best_fit`, or `next_fit`.
- `--any_sort_order=<order>`, where `<order>` is one of `cmdline` or `descending_size`.

Use `first_fit` when you want to fill regions in order.

Use `best_fit` when you want to fill regions to their maximum.

Use `worst_fit` when you want to fill regions evenly. With equal sized regions and sections `worst_fit` fills regions cyclically.

Use `next_fit` when you need a more deterministic fill pattern.

If the linker attempts to fill a region to its limit, as it does with `first_fit` and `best_fit`, it might overfill the region. This is because linker-generated content such as padding and veneers are not known until sections have been assigned to `.ANY` selectors. If this occurs you might see the following error:

```
Error: L6220E: Execution region <regionname> size (<size> bytes) exceeds limit (<limit> bytes).
```

The `--any_contingency` option prevents the linker from filling the region up to its maximum. It reserves a portion of the region's size for linker-generated content and fills this contingency area only if no other regions have space. It is enabled by default for the `first_fit` and `best_fit` algorithms, because they are most likely to exhibit this behavior.

10.11.3 Prioritizing the placement of unassigned sections

You can give a priority ordering when placing unassigned sections with multiple `.ANY` module selectors.

Procedure

To prioritize the order of multiple `.ANY` sections use the `.ANY<num>` selector, where `<num>` is a positive integer starting at zero.
The highest priority is given to the selector with the highest integer.

The following example shows how to use `.ANY<num>`:

```
lr1 0x8000 1024
{
```

```
    er1 +0 512
    {
        .ANY1(+RO) ; evenly distributed with er3
    }
    er2 +0 256
    {
        .ANY2(+RO) ; Highest priority, so filled first
    }
    er3 +0 256
    {
        .ANY1(+RO) ; evenly distributed with er1
    }
}
```

10.11.4 Specify the maximum region size permitted for placing unassigned sections

You can specify the maximum size in a region that `armlink` can fill with unassigned sections.

Use the execution region attribute `ANY_SIZE <max_size>` to specify the maximum size in a region that `armlink` can fill with unassigned sections.

Be aware of the following restrictions when using this keyword:

- `<max_size>` must be less than or equal to the region size.
- If you use `ANY_SIZE` on a region without a `.ANY` selector, it is ignored by `armlink`.

When `ANY_SIZE` is present, `armlink` does not attempt to calculate contingency and strictly follows the `.ANY` priorities.

When `ANY_SIZE` is not present for an execution region containing a `.ANY` selector, and you specify the `--any_contingency` command-line option, then `armlink` attempts to adjust the contingency for that execution region. The aims are to:

- Never overflow a `.ANY` region.
- Make sure there is a contingency reserved space left in the given execution region. This space is reserved for veneers and section padding.

If you specify `--any_contingency` on the command line, it is ignored for regions that have `ANY_SIZE` specified. It is used as normal for regions that do not have `ANY_SIZE` specified.

Example

The following example shows how to use `ANY_SIZE`:

```
LOAD_REGION 0x0 0x3000
{
    ER_1 0x0 ANY_SIZE 0xF00 0x1000
    {
        .ANY
    }
    ER_2 0x0 ANY_SIZE 0xFB0 0x1000
    {
        .ANY
    }
```

```
    ER_3 0x0 ANY_SIZE 0x1000 0x1000
    {
        .ANY
    }
}
```

In this example:

- `ER_1` has `0x100` reserved for linker-generated content.
- `ER_2` has `0x50` reserved for linker-generated content. That is about the same as the automatic contingency of `--any_contingency`.
- `ER_3` has no reserved space. Therefore, 100% of the region is filled, with no contingency for veneers. Omitting the `ANY_SIZE` parameter causes 98% of the region to be filled, with a two percent contingency for veneers.

10.11.5 Examples of using placement algorithms for .ANY sections

These examples show the operation of the placement algorithms for `RO-CODE` sections in `sections.o`.

The input section properties and ordering are shown in the following table:

Table 10-3: Input section properties for placement of .ANY sections

Name	Size (bytes)
sec1	0x4
sec2	0x4
sec3	0x4
sec4	0x4
sec5	0x4
sec6	0x4

The scatter file that the examples use is:

```
LR 0x100
{
  ER_1 0x100 0x10
  {
     .ANY
  }
  ER_2 0x200 0x10
  {
     .ANY
  }
}
```

Note

These examples have `--any_contingency` disabled.

Example for first_fit, next_fit, and best_fit

This example shows the image memory map where several sections of equal size are assigned to two regions with one selector. The selectors are equally specific, equivalent to `.ANY(+RO)` and have no priority.

```
Execution Region ER_1 (Base: 0x00000100, Size: 0x00000010, Max: 0x00000010,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000100   0x00000004   Code   RO            1    sec1                sections.o
0x00000104   0x00000004   Code   RO            2    sec2                sections.o
0x00000108   0x00000004   Code   RO            3    sec3                sections.o
0x0000010c   0x00000004   Code   RO            4    sec4                sections.o


Execution Region ER_2 (Base: 0x00000200, Size: 0x00000008, Max: 0x00000010,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000200   0x00000004   Code   RO            5    sec5                sections.o
0x00000204   0x00000004   Code   RO            6    sec6                sections.o
```

In this example:

- For `first_fit`, the linker first assigns all the sections it can to `ER_1`, then moves on to `ER_2` because that is the next available region.
- For `next_fit`, the linker does the same as `first_fit`. However, when `ER_1` is full it is marked as `FULL` and is not considered again. In this example, `ER_1` is full. `ER_2` is then considered.
- For `best_fit`, the linker assigns `sec1` to `ER_1`. It then has two regions of equal priority and specificity, but `ER_1` has less space remaining. Therefore, the linker assigns `sec2` to `ER_1`, and continues assigning sections until `ER_1` is full.

Example for worst_fit

This example shows the image memory map when using the `worst_fit` algorithm.

```
Execution Region ER_1 (Base: 0x00000100, Size: 0x0000000c, Max: 0x00000010,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000100   0x00000004   Code   RO            1    sec1                sections.o
0x00000104   0x00000004   Code   RO            3    sec3                sections.o
0x00000108   0x00000004   Code   RO            5    sec5                sections.o


Execution Region ER_2 (Base: 0x00000200, Size: 0x0000000c, Max: 0x00000010,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000200   0x00000004   Code   RO            2    sec2                sections.o
0x00000204   0x00000004   Code   RO            4    sec4                sections.o
0x00000208   0x00000004   Code   RO            6    sec6                sections.o
```

The linker first assigns `sec1` to `ER_1`. It then has two equally specific and priority regions. It assigns `sec2` to the one with the most free space, `ER_2` in this example. The regions now have the same

amount of space remaining, so the linker assigns `sec3` to the first one that appears in the scatter file, that is `ER_1`.


Note

The behavior of `worst_fit` is the default behavior in this version of the linker, and it is the only algorithm available in earlier linker versions.

10.11.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority

This example shows the operation of the `next_fit` placement algorithm for `RO-CODE` sections in `sections.o`.

The input section properties and ordering are shown in the following table:

Table 10-4: Input section properties for placement of sections with next_fit

Name	Size
sec1	0x14
sec2	0x14
sec3	0x10
sec4	0x4
sec5	0x4
sec6	0x4

The scatter file used for the examples is:

```
LR 0x100
{
  ER_1 0x100 0x20
  {
     .ANY1(+RO-CODE)
  }
  ER_2 0x200 0x20
  {
     .ANY2(+RO)
  }
  ER_3 0x300 0x20
  {
     .ANY3(+RO)
  }
}
```


Note

This example has `--any_contingency` disabled.

The `next_fit` algorithm is different to the others in that it never revisits a region that is considered to be full. This example also shows the interaction between priority and specificity of selectors. This is the same for all the algorithms.

```
Execution Region ER_1 (Base: 0x00000100, Size: 0x00000014, Max: 0x00000020,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000100   0x00000014   Code   RO            1    sec1                sections.o


Execution Region ER_2 (Base: 0x00000200, Size: 0x0000001c, Max: 0x00000020,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000200   0x00000010   Code   RO            3    sec3                sections.o
0x00000210   0x00000004   Code   RO            4    sec4                sections.o
0x00000214   0x00000004   Code   RO            5    sec5                sections.o
0x00000218   0x00000004   Code   RO            6    sec6                sections.o


Execution Region ER_3 (Base: 0x00000300, Size: 0x00000014, Max: 0x00000020,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000300   0x00000014   Code   RO            2    sec2                sections.o
```

In this example:

- The linker places `sec1` in `ER_1` because `ER_1` has the most specific selector. `ER_1` now has `0x6` bytes remaining.
- The linker then tries to place `sec2` in `ER_1`, because it has the most specific selector, but there is not enough space. Therefore, `ER_1` is marked as full and is not considered in subsequent placement steps. The linker chooses `ER_3` for `sec2` because it has higher priority than `ER_2`.
- The linker then tries to place `sec3` in `ER_3`. It does not fit, so `ER_3` is marked as full and the linker places `sec3` in `ER_2`.
- The linker now processes `sec4`. This is `0x4` bytes so it can fit in either `ER_1` or `ER_3`. Because both of these sections have previously been marked as full, they are not considered. The linker places all remaining sections in `ER_2`.
- If another section `sec7` of size `0x8` exists, and is processed after `sec6` the example fails to link. The algorithm does not attempt to place the section in `ER_1` or `ER_3` because they have previously been marked as full.

10.11.7 Examples of using sorting algorithms for .ANY sections

These examples show the operation of the sorting algorithms for `RO-CODE` sections in `sections_a.o` and `sections_b.o`.

The input section properties and ordering are shown in the following table:

sections_a.o		sections_b.o	
Name	Size	Name	Size
seca_1	0x4	secb_1	0x4
seca_2	0x4	secb_2	0x4
seca_3	0x10	secb_3	0x10
seca_4	0x14	secb_4	0x14

Descending size example

The following linker command-line options are used for this example:

```
--any_sort_order=descending_size sections_a.o sections_b.o --scatter scatter.txt
```

The following table shows the order that the sections are processed by the `.ANY` assignment algorithm.

Table 10-6: Sort order for descending_size algorithm

Name	Size
seca_4	0x14
secb_4	0x14
seca_3	0x10
secb_3	0x10
seca_1	0x4
seca_2	0x4
secb_1	0x4
secb_2	0x4

With `--any_sort_order=descending_size`, sections of the same size use the creation index as a tiebreaker.

Command-line example

The following linker command-line options are used for this example:

```
--any_sort_order=cmdline sections_a.o sections_b.o --scatter scatter.txt
```

The following table shows the order that the sections are processed by the `.ANY` assignment algorithm.

Table 10-7: Sort order for cmdline algorithm

Name	Size
seca_1	0x4
seca_2	0x4
seca_3	0x10
seca_4	0x14

Name	Size
secb_1	0x4
secb_2	0x4
secb_3	0x10
secb_4	0x14

That is, the input sections are sorted by command-line index.

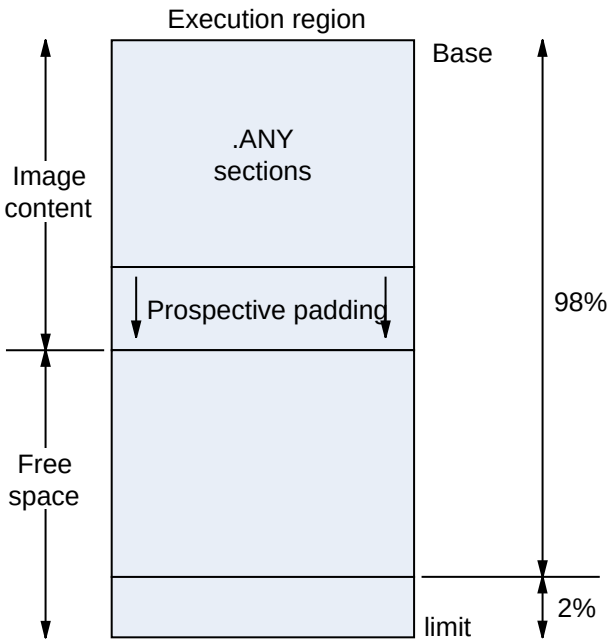
10.11.8 Behavior when .ANY sections overflow because of linker-generated content

Because linker-generated content might cause `.ANY` sections to overflow, a contingency algorithm is included in the linker.

The linker does not know the address of a section until it is assigned to a region. Therefore, when filling `.ANY` regions, the linker cannot calculate the contingency space and cannot determine if calling functions require veneers. The linker provides a contingency algorithm that gives a worst-case estimate for padding and an extra two percent for veneers. To enable this algorithm, use the `--any_contingency` command-line option.

The following diagram represents an example image layout during `.ANY` placement:

Figure 10-6: .ANY contingency



The downward arrows for prospective padding show that the prospective padding continues to grow as more sections are added to the `.ANY` selector.

Prospective padding is dealt with before the two percent veneer contingency.

When the prospective padding is cleared, the priority is set to zero. When the two percent is cleared, the priority is decremented again.

You can also use the `ANY_SIZE` keyword on an execution region to specify the maximum amount of space in the region to set aside for `.ANY` section assignments.

You can use the `armlink` command-line option `--info=any` to get extra information on where the linker has placed sections. This information can be useful when trying to debug problems.

Note

When there is only one `.ANY` selector, it might not behave identically to `*`. The algorithms that are used to determine the size of the section and place data still run with `.ANY` and they try to estimate the impact of changes that might affect the size of sections. These algorithms do not run if `*` is used instead. When it is appropriate to use one or the other of `.ANY` or `*`, then you must not use a single `.ANY` selector that applies to a kind of data, such as RO, RW, or ZI. For example, `.ANY (+RO)`.

You might see error `L6407E` generated, for example:

```
Error: L6407E: Sections of aggregate size 0x128 bytes could not fit
 into .ANY selector(s).
```

However, increasing the section size by `0x128` bytes does not necessarily lead to a successful link. The failure to link is because of the extra data, such as region table entries, that might end up in the region after adding more sections.

Example

1. Create the following `foo.c` program:

```
#include "stdio.h"

int array[10] __attribute__ ((section ("ARRAY")));

struct S {
    char A[8];
    char B[4];
};
struct S s;

struct S* get()
{
    return &s;
}

int sqr(int n1);

int gSquared __attribute__((section(".ARM.__at_0x5000")));  // Place at 0x5000

int sqr(int n1)
{
    return n1*n1;
}

int main(void) {
    int i;
    for (i=0; i<10; i++) {
        array[i]=i*i;
        printf("%d\n", array[i]);
    }
    gSquared=sqr(i);
    printf("%d squared is: %d\n", i, gSquared);

    return sizeof(array);
}
```

2. Create the following `scatter.scat` file:

```
LOAD_REGION 0x0 0x3000
{
   ER_1 0x0 0x1000
   {
      .ANY
   }
   ER_2 (ImageLimit(ER_1)) 0x1500
   {
      .ANY
   }
   ER_3 (ImageLimit(ER_2)) 0x500
```

```
    {
       .ANY
    }
    ER_4 (ImageLimit(ER_3)) 0x1000
    {
        *(+RW,+ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
```

3. Compile and link the program as follows:

```
armclang -c --target=arm-arm-none-eabi -mcpu=cortex-m4 -o foo.o foo.c
armlink --cpu=cortex-m4 --any_contingency --scatter=scatter.scat --info=any -o
 foo.axf foo.o
```

The following shows an example of the information generated:

```
==============================================================================


Sorting unassigned sections by descending size for .ANY placement.
Using Worst Fit .ANY placement algorithm.
.ANY contingency enabled.

Exec Region    Event                          Idx          Size        Section Name
                Object
ER_2           Assignment: Worst fit          144          0x0000041a  .text
                c_wu.l(_printf_fp_dec.o)
ER_2           Assignment: Worst fit          261          0x00000338  CL$
$btod_div_common           c_wu.l(btod.o)
ER_1           Assignment: Worst fit          146          0x000002fc  .text
                c_wu.l(_printf_fp_hex.o)
ER_2           Assignment: Worst fit          260          0x00000244  CL$
$btod_mult_common          c_wu.l(btod.o)
...
ER_1           Assignment: Worst fit          3            0x00000090  .text
                foo.o
...
ER_3           Assignment: Worst fit          100          0x0000000a
.ARM.Collect$$_printf_percent$$00000007    c_wu.l(_printf_ll.o)
ER_3           Info: .ANY limit reached       -            -           -
                -
ER_1           Assignment: Highest priority   423          0x0000000a  .text
                c_wu.l(defsig_exit.o)
...
.ANY contingency summary
Exec Region    Contingency     Type
ER_1           161             Auto
ER_2           180             Auto
ER_3           73              Auto

==============================================================================


Sorting unassigned sections by descending size for .ANY placement.
Using Worst Fit .ANY placement algorithm.
.ANY contingency enabled.

Exec Region    Event                          Idx          Size        Section Name
                Object
```

```
ER_2            Info: .ANY limit reached      -           -           -
                 -
ER_1            Info: .ANY limit reached      -           -           -
                 -
ER_3            Info: .ANY limit reached      -           -           -
                 -
ER_2            Assignment: Worst fit         533         0x00000034  !!!scatter
                 c_wu.l(__scatter.o)
ER_2            Assignment: Worst fit         535         0x0000001c  !!handler_zi
                 c_wu.l(__scatter_zi.o)
```

10.12 Placing veneers with a scatter file

You can place veneers at a specific location with a linker-generated symbol.

About this task

Veneers allow switching between A32 and T32 code or allow a longer program jump than can be specified in a single instruction.

Procedure

To place veneers at a specific location, include the linker-generated symbol `Veneer$$Code` in a scatter file. At most, one execution region in the scatter file can have the `*(Veneer$$Code)` section selector.
If it is safe to do so, the linker places veneer input sections into the region identified by the `*(Veneer$$Code)` section selector. It might not be possible for a veneer input section to be assigned to the region because of address range problems or execution region size limitations. If the veneer cannot be added to the specified region, it is added to the execution region containing the relocated input section that generated the veneer.

Note

Instances of `*(IWV$$Code)` in scatter files from earlier versions of Arm tools are automatically translated into `*(Veneer$$Code)`. Use `*(Veneer$$Code)` in new descriptions.

`*(Veneer$$Code)` is ignored when the amount of code in an execution region exceeds 4MB of 16-bit T32 code, 16MB of 32-bit T32 code, and 32MB of A32 code.

Note

There are no state-change veneers in A64.

10.13 Preprocessing a scatter file

You can pass a scatter file through a C preprocessor. This permits access to all the features of the C preprocessor.

Use the first line in the scatter file to specify a preprocessor command that the linker invokes to process the file. The command is of the form:

```
#! preprocessor [preprocessor_flags]
```

Most typically the command is of the form `#! armclang --target=<target> -march=<architecture> -E -x c`. This passes the scatter file through the `armclang` preprocessor.

You can:

- Add preprocessing directives to the top of the scatter file.
- Use simple expression evaluation in the scatter file.

For example, a scatter file, `file.scat`, might contain:

```
#! armclang --target=arm-arm-none-eabi -march=armv8-a -E -x c

#define ADDRESS 0x20000000
#include "include_file_1.h"

LR1 ADDRESS
{
    ...
}
```

The linker parses the preprocessed scatter file and treats the directives as comments.

You can also use the `--predefine` command-line option to assign values to constants. For this example:

1. Modify `file.scat` to delete the directive `#define ADDRESS 0x20000000`.
2. Specify the command:

```
armlink --predefine="-DADDRESS=0x20000000" --scatter=file.scat
```

Default behavior for armclang -E in a scatter file

`armlink` behaves in the same way as `armclang` when invoking other Arm tools.

`armlink` searches for the `armclang` binary in the following order:

1. The same location as `armlink`.
2. The `PATH` locations.

`armlink` invokes `armclang` with the `-I<scatter_file_path>` option so that any preprocessor directives with relative paths work. The linker only adds this option if the full name of the

preprocessor tool given is `armclang` or `armclang.exe`. This means that if an absolute path or a relative path is given, the linker does not give the `-I<scatter_file_path>` option to the preprocessor. This also happens with the `--cpu` option.

On Windows, `.exe` suffixes are handled, so `armclang.exe` is considered the same as `armclang`. Executable names are case insensitive, so `armclang` is considered the same as `armclang`. The portable way to write scatter file preprocessing lines is to use correct capitalization and omit the `.exe` suffix.

Use of other preprocessors in a scatter file

You must ensure that the preprocessing command line is appropriate for execution on the host system.

This means:

- The string must be correctly quoted for the host system. The portable way to do this is to use double-quotes.
- Single quotes and escaped characters are not supported and might not function correctly.
- The use of a double-quote character in a path name is not supported and might not work.

These rules also apply to any strings passed with the `--predefine` option.

All preprocessor executables must accept the `-o <file>` option to mean output to file and accept the input as a filename argument on the command line. These options are automatically added to the user command line by `armlink`. Any options to redirect preprocessing output in the user-specified command line are not supported.

10.14 Reserving an empty block of memory

You can reserve an empty block of memory with a scatter file, such as the area used for the stack.

To reserve an empty block of memory, add an execution region in the scatter file and assign the `EMPTY` attribute to that region.

10.14.1 Characteristics of a reserved empty block of memory

An empty block of memory that is reserved with a scatter-loading description has certain characteristics.

The block of memory does not form part of the load region, but is assigned for use at execution time. Because it is created as a dummy ZI region, the linker uses the following symbols to access it:

- `Image$$<region_name>$$ZI$$Base`.
- `Image$$<region_name>$$ZI$$Limit`.
- `Image$$<region_name>$$ZI$$Length`.

If the length is given as a negative value, the address is taken to be the end address of the region. This address must be an absolute address and not a relative one.

10.14.2 Example of reserving an empty block of memory

This example shows how to reserve and empty block of memory for stack and heap using a scatter-loading description. It also shows the related symbols that the linker generates.

In the following example, the execution region definition `STACK 0x800000 EMPTY -0x10000` defines a region that is called `STACK`. The region starts at address `0x7F0000` and ends at address `0x800000`:

```
LR_1 0x80000                          ; load region starts at 0x80000
{
    STACK 0x800000 EMPTY -0x10000     ; region ends at 0x800000 because of the
                                      ; negative length. The start of the region
                                      ; is calculated using the length.
    {
                                      ; Empty region for placing the stack
    }

    HEAP +0 EMPTY 0x10000             ; region starts at the end of previous
                                      ; region. End of region calculated using
                                      ; positive length
    {
                                      ; Empty region for placing the heap
    }
    ...                               ; rest of scatter-loading description
}
```

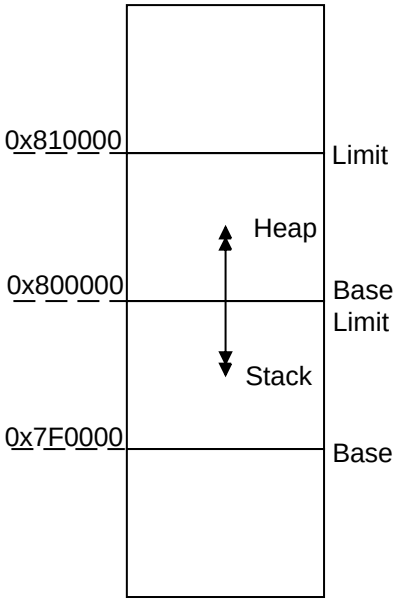
Note

The dummy ZI region that is created for an `EMPTY` execution region is not initialized to zero at runtime.

If the address is in relative (`+<offset>`) form and the length is negative, the linker generates an error.

The following figure shows a diagrammatic representation for this example.

Figure 10-7: Reserving a region for the stack



In this example, the linker generates the following symbols:

```
Image$$STACK$$ZI$$Base      = 0x7f0000
Image$$STACK$$ZI$$Limit     = 0x800000
Image$$STACK$$ZI$$Length    = 0x10000
Image$$HEAP$$ZI$$Base       = 0x800000
Image$$HEAP$$ZI$$Limit      = 0x810000
Image$$HEAP$$ZI$$Length     = 0x10000
```

Note

The `EMPTY` attribute applies only to an execution region. The linker generates a warning and ignores an `EMPTY` attribute that is used in a load region definition.

The linker checks that the address space used for the `EMPTY` region does not overlap any other execution region.

10.15 Alignment of regions to page boundaries

You can produce an ELF file with each execution region starting at a page boundary.

The linker provides the following built-in functions to help create load and execution regions on page boundaries:

- `AlignExpr`, to specify an address expression.
- `GetPageSize`, to obtain the page size for use in `AlignExpr`. If you use `GetPageSize`, you must also use the `--paged` linker command-line option.
- `SizeOfHeaders()`, to return the size of the ELF header and Program Header table.


Note

- Alignment on an execution region causes both the load address and execution address to be aligned.
- The default page size is `0x8000`. To change the page size, specify the `--pagesize` linker command-line option.

To produce an ELF file with each execution region starting on a new page, and with code starting on the next page boundary after the header information:

```
LR1 0x0 + SizeOfHeaders()
{
    ER_RO +0
    {
        *(+RO)
    }
    ER_RW AlignExpr(+0, GetPageSize())
    {
        *(+RW)
    }
    ER_ZI AlignExpr(+0, GetPageSize())
    {
        *(+ZI)
    }
}
```

If you set up your ELF file in this way, then you can memory-map it onto an operating system in such a way that:

- RO and RW data can be given different memory protections, because they are placed in separate pages.
- The load address everything expects to run at is related to its offset in the ELF file by specifying `SizeOfHeaders()` for the first load region.

10.16 Alignment of execution regions and input sections

There are situations when you want to align code and data sections. How you deal with them depends on whether you have access to the source code.

Aligning when it is convenient for you to modify the source and recompile

When it is convenient for you to modify the original source code, you can align at compile time with the `__align(n)` keyword, for example.

Aligning when it is not convenient for you to modify the source and recompile

It might not be convenient for you to modify the source code for various reasons. For example, your build process might link the same object file into several images with different alignment requirements.

When it is not convenient for you to modify the source code, then you must use the following alignment specifiers in a scatter file:

ALIGNALL

Increases the section alignment of all the sections in an execution region, for example:

```
ER_DATA ... ALIGNALL 8
{
    .. ;selectors
}
```

OVERALIGN

Increases the alignment of a specific section, for example:

```
ER_DATA ...
{
    *.o(.bar, OVERALIGN 8)
    ... ;selectors
}
```


Note

`armlink` does not increase the alignment of some sections where it might be unsafe to do so. For more information, see Syntax of an input section description.

11. Overlay support in Arm Compiler for Embedded FuSa 6

There are situations when you might want to load some code in memory, then replace it with different code. For example, your system might have memory constraints that mean you cannot load all code into memory at the same time.

The solution is to create an overlay region where each piece of overlaid code is unloaded and loaded by an overlay manager. Arm® Compiler for Embedded FuSa supports:

- An automatic overlay mechanism, where the linker decides how your code sections get allocated to overlay regions.
- A manual overlay mechanism, where you manually arrange the allocation of the code sections.


Note

Arm Compiler for Embedded FuSa does not support using both manual and automatic overlays within the same program.

11.1 Automatic overlay support

For the linker to automatically allocate code sections to overlay regions, you must modify your C or assembly code to identify the parts to be overlaid. You must also set up a scatter file to locate the overlays.


Note

Arm® Compiler for Embedded FuSa does not support using both manual and automatic overlays within the same program.

The automatic overlay mechanism consists of:

- Special section names that you can use in your object files to mark code as overlaid.
- The `AUTO_OVERLAY` execution region attribute. Use this in a scatter file to indicate regions of memory where the linker assigns the overlay sections for loading into at runtime.
- The command-line option `--overlay-veneers` to make the linker redirect calls between overlays to a veneer that lets an overlay manager unload and load the correct overlays.
- A set of data tables and symbol names provided by the linker that you can use to write the overlay manager.
- The `armlink` command-line option `--emit_debug_overlay_section` to add extra debug information to the image. This option permits an overlay-aware debugger to track which overlay is currently active.

Related information

Automatically placing code sections in overlay regions on page 227
Overlay veneer on page 228
Overlay data tables on page 229
Limitations of automatic overlay support on page 230
About writing an overlay manager for automatically placed overlays on page 231

11.1.1 Automatically placing code sections in overlay regions

Arm® Compiler for Embedded FuSa can automatically place code sections into overlay regions.

About this task

You identify the sections in your code that are to become overlays by giving them names of the form `.ARM.overlay<N>`, where `<N>` is an integer identifier. You then use a scatter file to indicate those regions of memory where `armlink` is to assign the overlays for loading at runtime.

Each overlay region corresponds to an execution region that has the attribute `AUTO_OVERLAY` assigned in the scatter file. `armlink` allocates one set of integer identifiers to each of these overlay regions. It allocates another set of integer identifiers to each overlaid section with the name `.ARM.overlay<N>` that is defined in the object files.


Note

The numbers that are assigned to the overlay sections in your object files do not match up to the numbers that you put in the `.ARM.overlay<N>` section names.

Procedure

1. Declare the functions that you want the `armlink` automatic overlay mechanism to process.
   - In C, use a function attribute, for example:

```
__attribute__((section(".ARM.overlay1"))) void foo(void) { ... }

__attribute__((section(".ARM.overlay2"))) void bar(void) { ... }
```

   - In the `armclang` integrated assembler syntax, use the `.section` directive, for example:

```
    .section    .ARM.overlay1,"ax",%progbits
    .global foo
    .p2align 2
    .type    foo,%function
foo:
    ...
    .fnend

    .section .ARM.overlay2,"ax",%progbits
    .global bar
    .p2align    2
    .type    bar,%function
bar:
    ...
```

```
    .fnend
```

- In `armasm` assembler syntax, use the `AREA` directive, for example:

```
    AREA |.ARM.overlay1|,CODE
foo PROC
    ...
    ENDP

    AREA |.ARM.overlay2|,CODE
bar PROC
    ...
    ENDP
```

Note

You can only overlay code sections. Data sections must never be overlaid.

2. Specify the locations to load the code sections from and to in a scatter file. Use the `AUTO_OVERLAY` keyword on one or more execution regions.
   The execution regions must not have any section selectors. For example:

```
OVERLAY_LOAD_REGION 0x10000000
{
    OVERLAY_EXECUTE_REGION_A 0x20000000 AUTO_OVERLAY 0x10000 { }
    OVERLAY_EXECUTE_REGION_B 0x20010000 AUTO_OVERLAY 0x10000 { }
}
```

   In this example, `armlink` emits a program header table entry that loads all the overlay data starting at address `0x10000000`. Also, each overlay is relocated so that it runs correctly if copied to address `0x20000000` or `0x20010000`. `armlink` chooses one of these addresses for each overlay.
3. When linking, specify the `--overlay_veneers` command-line option. This option causes `armlink` to arrange function calls between two overlays, or between non-overlaid code and an overlay, to be diverted through the entry point of an overlay manager.
   To permit an overlay-aware debugger to track the overlay that is active, specify the `--emit_debug_overlay_section` command-line option.

Related information

__attribute__((section("name"))) function attribute
AREA directive
Execution region attributes
--emit_debug_overlay_section linker option
--overlay_veneers linker option

11.1.2 Overlay veneer

`armlink` can generate an overlay veneer for each function call between two overlays, or between non-overlaid code and an overlay.

A function call or return can transfer control between two overlays or between non-overlaid code and an overlay. If the target function is not already present at its intended execution address, then the target overlay has to be loaded.

To detect whether the target overlay is present, `armlink` can arrange for all such function calls to be diverted through the overlay manager entry point, `__ARM_overlay_entry`. To enable this feature, use the `armlink` command-line option `--overlay_veneers`. This option causes a veneer to be generated for each affected function call, so that the call instruction, typically a `BL` instruction, points at the veneer instead of the target function. The veneer in turn saves some registers on the stack, loads some information about the target function and the overlay that it is in, and transfers control to the overlay manager entry point. The overlay manager must then:

- Ensure that the correct overlay is loaded and then transfer control to the target function.
- Restore the stack and registers to the state they were left in by the original `BL` instruction.
- If the function call originated inside an overlay, make sure that returning from the called function reloads the overlay being returned to.

Related information

--overlay_veneers linker option

11.1.3 Overlay data tables

`armlink` provides various symbols that point to a piece of read-only data, mostly arrays. This data describes the collection of overlays and overlay regions in the image.

The symbols are:

`Region$$Table$$AutoOverlay`

This symbol points to an array containing two 32-bit pointers per overlay region. For each region, the two pointers give the start address and end address of the overlay region. The start address is the first byte in the region. The end address is the first byte beyond the end of the region. The overlay manager can use this symbol to identify when the return address of a calling function is in an overlay region. In this case, a return thunk might be required.


Note

The regions are always sorted in ascending order of start address.

`Region$$Count$$AutoOverlay`

This symbol points to a single 16-bit integer (an unsigned short) giving the total number of overlay regions. That is, the number of entries in the arrays `Region$$Table$$AutoOverlay` and `CurrLoad$$Table$$AutoOverlay`.

`Overlay$$Map$$AutoOverlay`

This symbol points to an array containing a 16-bit integer (an unsigned short) per overlay. For each overlay, this table indicates which overlay region the overlay expects to be loaded into to run correctly.

`Size$$Table$$AutoOverlay`

This symbol points to an array containing a 32-bit word per overlay. For each overlay, this table gives the exact size of the data for the overlay. This size might be less than the size of its containing overlay region, because overlays typically do not fill their regions exactly.

In addition to the read-only tables, `armlink` also provides one piece of read/write memory:

`CurrLoad$$Table$$AutoOverlay`

This symbol points to an array containing a 16-bit integer (an unsigned short) for each overlay region. The array is intended for the overlay manager to store the identifier of the currently loaded overlay in each region. The overlay manager can then avoid reloading an already-loaded overlay.

All these data tables are optional. If your code does not refer to any particular table, then it is omitted from the image.

Related information

Automatic overlay support on page 226

11.1.4 Limitations of automatic overlay support

There are some limitations when using the automatic overlay feature.

The following limitations apply:

- The automatic overlay feature does not support C++.
- Even if you assign multiple functions to the same named section `.ARM.overlay<N>`, `armlink` still treats them as different overlays. `armlink` assigns a different integer ID to each overlay.
- The `armlink` command-line option `--any_placement` is ignored for the automatic overlay sections.
- The overlay system automatically generates veneers for direct calls between overlays, and between non-overlaid code and overlaid code. It automatically arranges that indirect calls through function pointers to functions in overlays work. However, if you pass a pointer to a non-overlaid function into an overlay that calls it, `armlink` has no way to insert a call to the overlay veneer. Therefore, the overlay manager has no opportunity to arrange to reload the overlay on behalf of the calling function on return.

In simple cases, this can still work. However, if the non-overlaid function calls something in a second overlay that conflicts with the overlay of its calling function, then a runtime failure occurs. For example:

```
__attribute__((section(".ARM.overlay1"))) void innermost(void)
{
    // do something
}

void non_overlaid(void)
{
    innermost();
}

typedef void (*function_pointer)(void);

__attribute__((section(".ARM.overlay2"))) void call_via_ptr(function_pointer f)
{
    f();
}

int main(void)
{
    // Call the overlaid function call_via_ptr() and pass it a pointer
    // to non_overlaid(). non_overlaid() then calls the function
    // innermost() in another overlay. If call_via_ptr() and innermost()
    // are allocated to the same overlay region by the linker, then there
    // is no way for call_via_ptr to have been reloaded by the time control
    // has to return to it from non_overlaid().

    call_via_ptr(non_overlaid);
}
```

Related information

Automatic overlay support on page 226

11.1.5 About writing an overlay manager for automatically placed overlays

To write an overlay manager to handle loading and unloading of overlays, you must provide an implementation of the overlay manager entry point.

The overlay manager entry point `__ARM_overlay_entry` is the location that the linker-generated veneers expect to jump to. The linker also provides some tables of data to enable the overlay manager to find the overlays and the overlay regions to load.

The entry point is called by the linker overlay veneers as follows:

- r0 contains the integer identifier of the overlay containing the target function.
- r1 contains the execution address of the target function. That is, the address that the function appears at when its overlay is loaded.
- The overlay veneer pushes six 32-bit words onto the stack. These words comprise the values of the r0, r1, r2, r3, r12, and lr registers of the calling function. If the call instruction is a `BL`, the value of lr is the one written into lr by the `BL` instruction, not the one before the `BL`.

The overlay manager has to:

1. Load the target overlay.
2. Restore all six of the registers from the stack.
3. Transfer control to the address of the target function that is passed in r1.

The overlay manager might also have to modify the value it passes to the calling function in lr to point at a return thunk routine. This routine would reload the overlay of the calling function and then return control to the original value of the lr of the calling function.

There is no sensible place already available to store the original value of lr for the return thunk to use. For example, there is nowhere on the stack that can contain the value. Therefore, the overlay manager has to maintain its own stack-organized data structure. The data structure contains the saved lr value and the corresponding overlay ID for each time the overlay manager substitutes a return thunk during a function call, and keeps it synchronized with the main call stack.


Note

Because this extra parallel stack has to be maintained, then you cannot use stack manipulations unless it is customized to keep the parallel stack of the overlay manager consistent. Some examples of stack manipulations include cooperative or preemptive thread switching, coroutines, and the `setjmp()` and `longjmp()` functions.

The `armlink` option `--info=auto_overlay` causes the linker to write out a text summary of the overlays in the image it outputs. The summary consists of the integer ID, start address, and size of each overlay. You can use this information to extract the overlays from the image, for example from the output of the `fromelf` option `--bin`. You can then put them in a separate peripheral storage system. Therefore, you still know which chunk of data goes with which overlay ID when you have to load one of them in the overlay manager.

Related information

Automatic overlay support on page 226
--info linker option

11.2 Manual overlay support

To manually allocate code sections to overlay regions, you must set up a scatter file to locate the overlays.


Note

Arm® Compiler for Embedded FuSa does not support using both manual and automatic overlays within the same program.

The manual overlay mechanism consists of:

- The OVERLAY attribute for load regions and execution regions. Use this attribute in a scatter file to indicate regions of memory where the linker assigns the overlay sections for loading into at runtime.
- The following `armlink` command-line options to add extra debug information to the image:
  - `--emit_debug_overlay_relocs`.
  - `--emit_debug_overlay_section`.

  This extra debug information permits an overlay-aware debugger to track which overlay is active.

Related information

Manually placing code sections in overlay regions on page 233
Writing an overlay manager for manually placed overlays on page 235

11.2.1 Manually placing code sections in overlay regions

You can place multiple execution regions at the same address with overlays.

The OVERLAY attribute allows you to place multiple execution regions at the same address. An overlay manager is required to make sure that only one execution region is instantiated at a time. Arm® Compiler for Embedded FuSa does not provide an overlay manager.

The following example shows the definition of a static section in RAM followed by a series of overlays. Here, only one of these sections is instantiated at a time.

```
EMB_APP 0x8000
{
    ...
    STATIC_RAM 0x0                  ; contains most of the RW and ZI code/data
    {
        * (+RW,+ZI)
    }
    OVERLAY_A_RAM 0x1000 OVERLAY    ; start address of overlay...
    {
        module1.o (+RW,+ZI)
    }
    OVERLAY_B_RAM 0x1000 OVERLAY
    {
        module2.o (+RW,+ZI)
    }
    ...                         ; rest of scatter-loading description
}
```

The C library at startup does not initialize a region that is marked as OVERLAY. The contents of the memory that is used by the overlay region is the responsibility of an overlay manager. If the region contains initialized data, use the NOCOMPRESS attribute to prevent RW data compression.

You can use the linker defined symbols to obtain the addresses that are required to copy the code and data.

You can use the `OVERLAY` attribute on a single region that is not at the same address as a different region. Therefore, you can use an overlay region as a method to prevent the initialization of particular regions by the C library startup code. As with any overlay region, you must manually initialize them in your code.

An overlay region can have a relative base. The behavior of an overlay region with a `+<offset>` base address depends on the regions that precede it and the value of `+<offset>`. If they have the same `+<offset>` value, the linker places consecutive `+<offset>` regions at the same base address.

When a `+<offset>` execution region ER follows a contiguous overlapping block of overlay execution regions the base address of ER is:

```
limit address of the overlapping block of overlay execution regions + <offset>
```

The following table shows the effect of `+<offset>` when used with the `OVERLAY` attribute. `REGION1` appears immediately before `REGION2` in the scatter file:

Table 11-1: Using relative offset in overlays

REGION1 is set with `OVERLAY`	`+<offset>`	REGION2 Base Address
NO	`<offset>`	`REGION1 Limit + <offset>`
YES	`+0`	`REGION1 Base Address`
YES	`<non-zero offset>`	`REGION1 Limit + <non-zero offset>`

The following example shows the use of relative offsets with overlays and the effect on execution region addresses:

```
EMB_APP 0x8000
{
    CODE 0x8000
    {
        *(+RO)
    }
    # REGION1 Base = CODE limit
    REGION1 +0 OVERLAY
    {
        module1.o(*)
    }
    # REGION2 Base = REGION1 Base
    REGION2 +0 OVERLAY
    {
        module2.o(*)
    }
    # REGION3 Base = REGION2 Base = REGION1 Base
    REGION3 +0 OVERLAY
    {
        module3.o(*)
    }
    # REGION4 Base = REGION3 Limit + 4
    Region4 +4 OVERLAY
    {
        module4.o(*)
    }
}
```

If the length of the non-overlay area is unknown, you can use a zero relative offset to specify the start address of an overlay so that it is placed immediately after the end of the static section.

Related information

Load region descriptions
Load region attributes
Inheritance rules for load region address attributes
Considerations when using a relative address +offset for a load region
Considerations when using a relative address +offset for execution regions
--emit_debug_overlay_relocs linker option
--emit_debug_overlay_section linker option
ABI for the Arm Architecture: Support for Debugging Overlaid Programs

11.2.2 Writing an overlay manager for manually placed overlays

Overlays are not automatically copied to their runtime location when a function within the overlay is called. Therefore, you must write an overlay manager to copy overlays.

About this task

An overlay manager copies the required overlay to its execution address, and records the overlay that is in use at any one time. The overlay manager runs throughout the application, and is called whenever overlay loading is required. For instance, the overlay manager can be called before every function call that might require a different overlay segment to be loaded.

The overlay manager must ensure that the correct overlay segment is loaded before calling any function in that segment. If a function from one overlay is called while a different overlay is loaded, then some kind of runtime failure occurs. If such a failure is a possibility, the linker and compiler do not warn you because it is not statically determinable. The same is true for a data overlay.

The central component of this overlay manager is a routine to copy code and data from the load address to the execution address. This routine is based around the following linker defined symbols:

- `Load$$execution_region_name$$Base`, the load address.
- `Image$$execution_region_name$$Base`, the execution address.
- `Image$$execution_region_name$$Length`, the length of the execution region.

The implementation of the overlay manager depends on the system requirements. This procedure shows a simple method of implementing an overlay manager.

The copy routine that is called `load_overlay()` is implemented in `overlay_manager.c`. The routine uses `memcpy()` and `memset()` functions to copy CODE and RW data overlays, and to clear ZI data overlays.


Note

For RW data overlays, it is necessary to disable RW data compression for the whole project. You can disable compression with the linker command-line option `--datacompressor off`, or you can mark the execution region with the attribute `NOCOMPRESS`.

The assembly file `overlay_list.s` lists all the required symbols. This file defines and exports two common base addresses and a RAM space that is mapped to the overlay structure table:

```
code_base
data_base
overlay_regions
```

As specified in the scatter file, `armlink` places the two functions, `func1()` and `func2()`, and their corresponding data in `CODE_ONE`, `CODE_TWO`, `DATA_ONE`, and `DATA_TWO` regions, respectively. `armlink` has a special mechanism for replacing calls to functions with stubs. To use this mechanism, write a small stub for each function in the overlay that might be called from outside the overlay.

In this example, two stub functions `$Sub$$func1()` and `$Sub$$func2()` are created for the two functions `func1()` and `func2()` in `overlay_stubs.c`. These stubs call the overlay-loading function `load_overlay()` to load the corresponding overlay. After the overlay manager finishes its overlay loading task, the stub function can then call `$Super$$func1` to call the loaded function `func1()` in the overlay.

Procedure

1. Create the `overlay_manager.c` program to copy the correct overlay to the runtime addresses.

```
/* overlay_manager.c
 * Basic overlay manager
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Number of overlays present */
#define NUM_OVERLAYS 2

/* struct to hold addresses and lengths */
typedef struct overlay_region_t_struct
{
  void* load_ro_base;
  void* load_rw_base;
  void* exec_zi_base;
  unsigned int ro_length;
  unsigned int zi_length;
} overlay_region_t;

/* Record for current overlay */
int current_overlay = 0;

/* Array describing the overlays */
extern const overlay_region_t overlay_regions[NUM_OVERLAYS];

/* execution bases of the overlay regions - defined in overlay_list.s */
extern void * const code_base;
extern void * const data_base;

void load_overlay(int n)
{
  const overlay_region_t * selected_region;

  if(n == current_overlay)
  {
    printf("Overlay %d already loaded.\n", n);
    return;
  }
```

```
  /* boundary check */
  if(n<1 || n>NUM_OVERLAYS)
  {
    printf("Error - invalid overlay number %d specified\n", n);
    exit(1);
  }

  /* Load the corresponding overlay */
  printf("Loading overlay %d...\n", n);

  /* set selected region */
  selected_region = &overlay_regions[n-1];

  /* load code overlay */
  memcpy(code_base, selected_region->load_ro_base, selected_region->ro_length);

  /* load data overlay */
  memcpy(data_base, selected_region->load_rw_base,
         (unsigned int)selected_region->exec_zi_base - (unsigned int)data_base);

  /* Comment out the next line if your overlays have any static ZI variables
   * and should not be reinitialized each time, and move them out of the
   * overlay region in your scatter file */
  memset(selected_region->exec_zi_base, 0, selected_region->zi_length);

  /* update record of current overlay */
  current_overlay=n;

  printf("...Done.\n");

}
```

2. Create a separate source file for each of the functions, `func1.c` for `func1()` and `func2.c` for `func2()`.

```
// func1.c
#include <stdio.h>
#include <stdlib.h>

extern void foo(int x);

// Some RW and ZI data
char* func1_string = "func1 called\n";
int func1_values[20];

void func1(void)
{
  unsigned int i;
  printf("%s\n", func1_string);
  for(i = 19; i; i--)
  {
    func1_values[i] = rand();
    foo(i);
    printf("%d ", func1_values[i]);
  }
  printf("\n");
}
```

```
// func2.c
#include <stdio.h>

extern void foo(int x);

// Some RW and ZI data
char* func2_string = "func2 called\n";
int func2_values[10];

void func2(void)
```

```
{
  printf("%s\n", func2_string);
  foo(func2_values[9]);
}
```

3. Create the `main.c` program to demonstrate the overlay mechanism.

```
// main.c
#include <stdio.h>


/* Functions provided by the overlays */
extern void func1(void);
extern void func2(void);

int main(void)
{
  printf("Start of main()...\n");
  func1();
  func2();

  /*
   * Call func2() again to demonstrate that we don't need to
   * reload the overlay
   */
  func2();

  func1();
  printf("End of main()...\n");

  return 0;
}

void foo(int x)
{
    return;
}
```

4. Create `overlay_stubs.c` to provide two stub functions `$Sub$$func1()` and `$Sub$$func2()` for the two functions `func1()` and `func2()`.

```
// overlay_stub.c
extern void $Super$$func1(void);
extern void $Super$$func2(void);

extern void load_overlay(int n);

void $Sub$$func1(void)
{
  load_overlay(1);
  $Super$$func1();
}

void $Sub$$func2(void)
{
  load_overlay(2);
  $Super$$func2();
}
```

5. Create `overlay_list.s` that lists all the required symbols.

```
; overlay_list.s
    AREA    overlay_list, DATA, READONLY

    ; Linker-defined symbols to use

    IMPORT ||Load$$CODE_ONE$$Base||
    IMPORT ||Load$$CODE_TWO$$Base||
    IMPORT ||Load$$DATA_ONE$$Base||
    IMPORT ||Load$$DATA_TWO$$Base||
```

```
    IMPORT ||Image$$CODE_ONE$$Base||
    IMPORT ||Image$$DATA_ONE$$Base||
    IMPORT ||Image$$DATA_ONE$$ZI$$Base||
    IMPORT ||Image$$DATA_TWO$$ZI$$Base||

    IMPORT ||Image$$CODE_ONE$$Length||
    IMPORT ||Image$$CODE_TWO$$Length||

    IMPORT ||Image$$DATA_ONE$$ZI$$Length||
    IMPORT ||Image$$DATA_TWO$$ZI$$Length||

    ; Symbols to export

    EXPORT code_base
    EXPORT data_base
    EXPORT overlay_regions

; Common base execution addresses of the two OVERLAY regions

code_base DCD ||Image$$CODE_ONE$$Base||
data_base DCD ||Image$$DATA_ONE$$Base||

; Array of details for each region -
; see overlay_manager.c for structure layout

overlay_regions
; overlay 1
    DCD ||Load$$CODE_ONE$$Base||
    DCD ||Load$$DATA_ONE$$Base||
    DCD ||Image$$DATA_ONE$$ZI$$Base||
    DCD ||Image$$CODE_ONE$$Length||
    DCD ||Image$$DATA_ONE$$ZI$$Length||

; overlay 2
    DCD ||Load$$CODE_TWO$$Base||
    DCD ||Load$$DATA_TWO$$Base||
    DCD ||Image$$DATA_TWO$$ZI$$Base||
    DCD ||Image$$CODE_TWO$$Length||
    DCD ||Image$$DATA_TWO$$ZI$$Length||

    END
```

6. Create `retarget.c` to retarget the `__user_initial_stackheap` function.

```
// retarget.c
#include <rt_misc.h>

extern unsigned int Image$$HEAP$$ZI$$Base;
extern unsigned int Image$$STACKS$$ZI$$Limit;

__value_in_regs struct __initial_stackheap __user_initial_stackheap(
        unsigned R0, unsigned SP, unsigned R2, unsigned SL)
{
  struct __initial_stackheap config;

  config.heap_base = (unsigned int)&Image$$HEAP$$ZI$$Base;
  config.stack_base = (unsigned int)&Image$$STACKS$$ZI$$Limit;

  return config;
}
```

7. Create the scatter file, `embedded_scat.scat`.

```
; embedded_scat.scat
;;; Copyright Arm Limited 2002. All rights reserved.

;; Embedded scatter file

ROM_LOAD 0x24000000 0x04000000
```

```
{
    ROM_EXEC 0x24000000 0x04000000
    {
        * (InRoot$$Sections)      ; All library sections that must be in a root
 region
                                  ; e.g. __main.o, __scatter*.o, * (Region$
$Table)
        * (+RO)                   ; All other code
    }

    RAM_EXEC 0x10000
    {
        * (+RW, +ZI)
    }

    HEAP +0 EMPTY 0x3000
    {
    }

    STACKS 0x20000 EMPTY -0x3000
    {
    }

    CODE_ONE 0x08400000 OVERLAY 0x4000
    {
        overlay_one.o (+RO)
    }

    CODE_TWO 0x08400000 OVERLAY 0x4000
    {
        overlay_two.o (+RO)
    }

    DATA_ONE 0x08700000 OVERLAY 0x4000
    {
        overlay_one.o (+RW,+ZI)
    }

    DATA_TWO 0x08700000 OVERLAY 0x4000
    {
        overlay_two.o (+RW,+ZI)
    }

}
```

8. Build the example application:

```
armclang -c -g -target arm-arm-none-eabi -mcpu=cortex-a9 -O0 main.c
 overlay_stubs.c overlay_manager.c retarget.c
armclang -c -g -target arm-arm-none-eabi -mcpu=cortex-a9 -O0 func1.c -o
 overlay_one.o
armclang -c -g -target arm-arm-none-eabi -mcpu=cortex-a9 -O0 func2.c -o
 overlay_two.o
armasm --debug --cpu=cortex-a9 --keep overlay_list.s
armlink --cpu=cortex-a9 --datacompressor=off --scatter embedded_scat.scat main.o
 overlay_one.o overlay_two.o overlay_stubs.o overlay_manager.o overlay_list.o
 retarget.o -o image.axf
```

Related information

Manual overlay support on page 232
Use of $Super$$ and $Sub$$ to patch symbol definitions

12. Embedded Software Development

When developing embedded applications, the resources available in the development environment normally differ from the resources on the target hardware.

It is important to consider the process for moving an embedded application from the development or debugging environment to a system that runs standalone on target hardware.

When developing embedded software, you must consider the following:

- Understand the default compilation tool behavior and the target environment. You can then understand the steps that are necessary to move from a debug or development build to a standalone production version of the application.
- Some C library functionality executes by using debug environment resources. If used, you must reimplement this functionality to use target hardware.
- The toolchain has no knowledge of the memory map of any given target. You must tailor the image memory map to the memory layout of the target hardware.
- An embedded application must perform some initialization, such as stack and heap initialization, before the main application can be run. A complete initialization sequence requires code that you implement in addition to the Arm® Compiler for Embedded FuSa C library initialization routines.

12.1 Default compilation tool behavior

It is useful to be aware of the default behavior of the compilation tools if you do not yet know the full technical specifications of the target hardware.

For example, when you start work on software for an embedded application, you might not know the details of target peripheral devices, the memory map, or even the processor itself.

To enable you to proceed with software development before such details are known, the compilation tools have a default behavior that enables you to start building and debugging application code immediately.

In the Arm C library, support for some ISO C functionality, for example program I/O, can be provided by the host debugging environment. The mechanism that provides this functionality is known as semihosting. When semihosting is executed, the debug agent suspends program execution. The debug agent then uses the debug capabilities of the host (for example `printf` output to the debugger console) to service the semihosting operation before code execution is resumed on the target. The task performed by the host is transparent to the program running on the target.

Related information

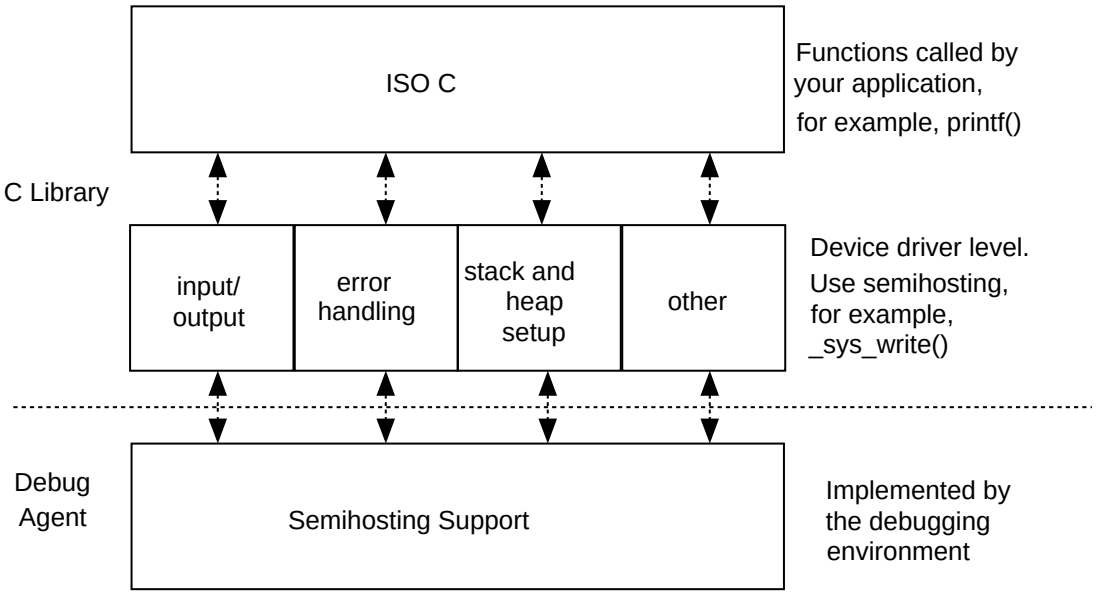
Semihosting for AArch32 and AArch64

12.2 C library structure

Conceptually, the C library can be divided into functions that are part of the ISO C standard, for example `printf()`, and functions that provide support to the ISO C standard.

For example, the following figure shows the C library implementing the function `printf()` by writing to the debugger console window. This implementation is provided by calling `_sys_write()`, a support function that executes a semihosting call, resulting in the default behavior using the debugger instead of target peripherals.

Figure 12-1: C library structure



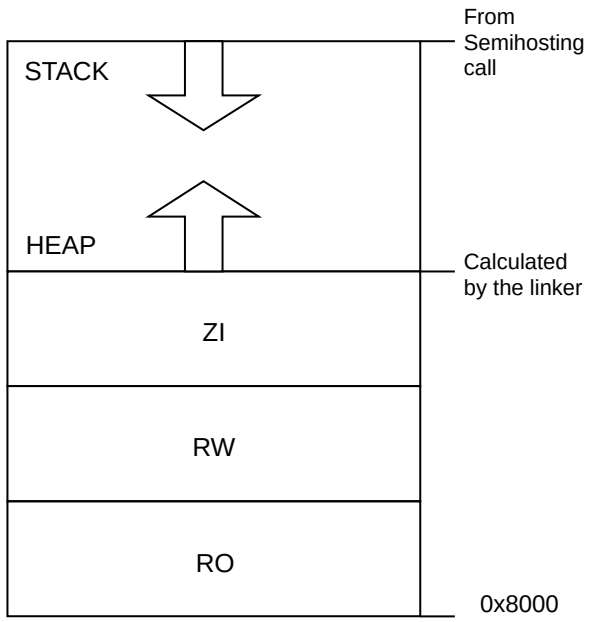
Related information

The Arm C and C++ libraries
The C and C++ library functions
Semihosting for AArch32 and AArch64

12.3 Default memory map

In an image where you have not described the memory map, the linker places code and data according to a default memory map.

Figure 12-2: Default memory map



Note

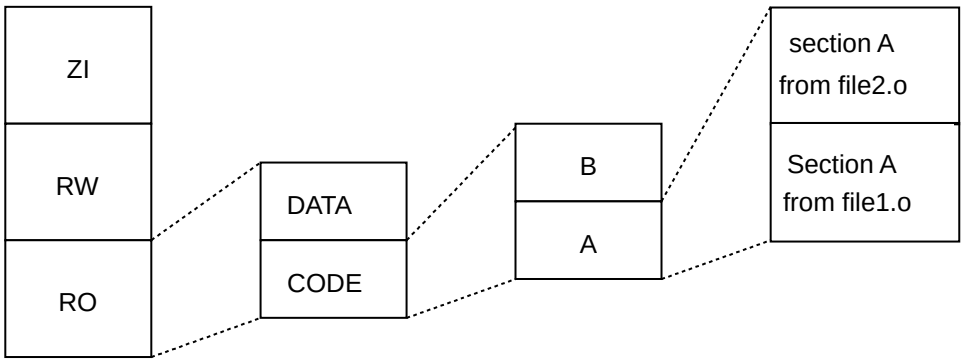
Processors that are based on Arm®v6-M and Armv7-M architectures have fixed memory maps. Having fixed memory maps makes porting software easier between different systems that are based on these processors.

The default memory map is described as follows:

- The image is linked to load and run at address `0x8000`. All read-only (RO) sections are placed first, followed by read/write (RW) sections, then zero-initialized (ZI) sections.
- The heap follows directly on from the top of ZI, so the exact location is decided at link time.
- The stack base location is provided by a semihosting operation during application startup. The value that this semihosting operation returns depends on the debug environment.

The linker observes a set of rules to decide where in memory code and data are located:

Figure 12-3: Linker placement rules



Generally, the linker sorts the Input sections by attribute (RO, RW, ZI), by name, and then by position in the input list.

To fully control the placement of code and data, you must use the scatter-loading mechanism.

Related information

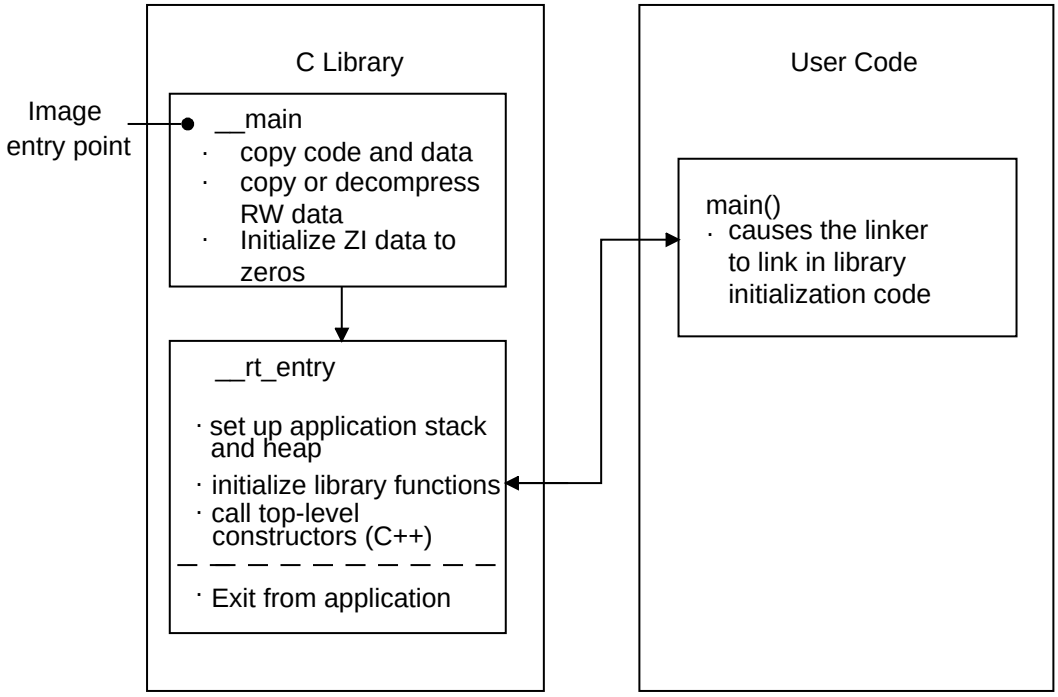
Tailoring the C library to your target hardware on page 245
The image structure
Section placement with the linker
About scatter-loading
Scatter file syntax
Cortex-M1 Technical Reference Manual
Cortex-M3 Technical Reference Manual
Semihosting for AArch32 and AArch64

12.4 Application startup

In most embedded systems, an initialization sequence executes to set up the system before the main task is executed.

The following figure shows the default initialization sequence.

Figure 12-4: Default initialization sequence



`__main` is responsible for setting up the memory and `__rt_entry` is responsible for setting up the run-time environment.

`__main` performs code and data copying, decompression, and zero initialization of the ZI data. It then branches to `__rt_entry` to set up the stack and heap, initialize the library functions and static data, and call any top level C++ constructors. `__rt_entry` then branches to `main()`, the entry to your application. When the main application has finished executing, `__rt_entry` shuts down the library, then hands control back to the debugger.

The function label `main()` has a special significance. The presence of a `main()` function forces the linker to link in the initialization code in `__main` and `__rt_entry`. Without a function labeled `main()`, the initialization sequence is not linked in, and as a result, some standard C library functionality is not supported.

Related information

--startup=symbol, --no_startup (armlink)
Arm Compiler C Library Startup and Initialization

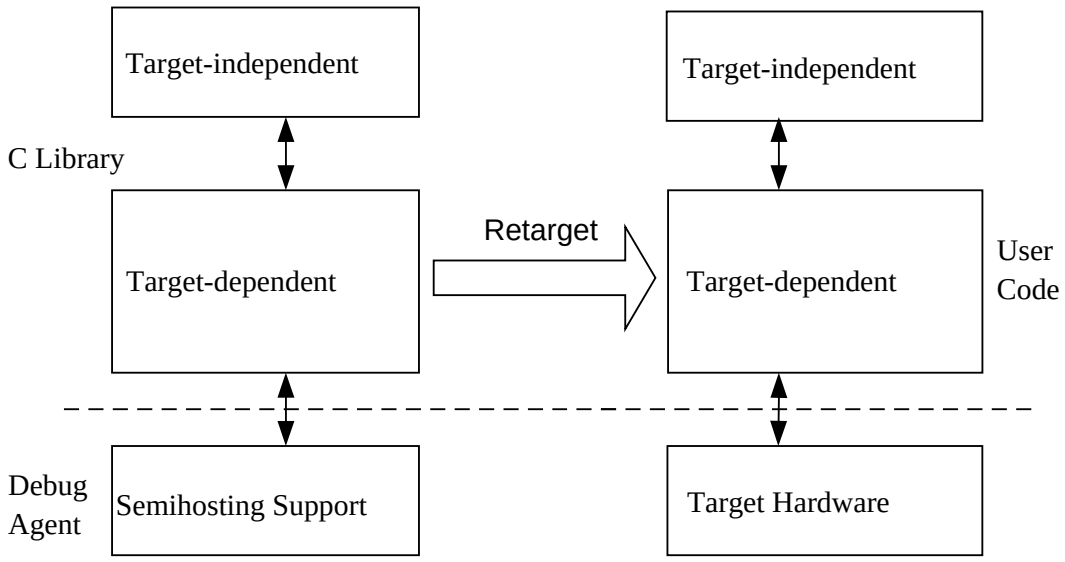
12.5 Tailoring the C library to your target hardware

You can provide your own implementations of C library functions to override the default behavior.

By default, the C library uses semihosting to provide device driver level functionality, enabling a host computer to act as an input and an output device. This functionality is useful because development hardware often does not have all the input and output facilities of the final system.

You can provide your own implementation of target-dependent C library functions to use target hardware. Your implementations are automatically linked in to your image instead of the C library implementations. The following figure shows this process, which is known as retargeting the C library.

Figure 12-5: Retargeting the C library



For example, you have a peripheral I/O device, such as an LCD screen. For this device, you want to override the library implementation of `fputc()`, which writes to the debugger console, with one that prints to the LCD. Because this implementation of `fputc()` is linked in to the final image, the entire `printf()` family of functions prints to the LCD.

Example implementation of `fputc()`

In this example, `fputc()` redirects the input character parameter to a serial output function `sendchar()`. `fputc()` assumes that `sendchar()` is implemented in a separate source file. In this way,

`fputc()` acts as an abstraction layer between target-dependent output and the C library standard output functions.

```
extern void sendchar(char *ch);
int fputc(int ch, FILE *f)
{   /* e.g. write a character to an LCD screen */
    char tempch = ch;
    sendchar(&tempch);
    return ch;
}
```

In a standalone application, you are unlikely to support semihosting operations. Therefore, you must remove all calls to target-dependent C library functions or reimplement them with non-semihosting functions.

Related information

Using the libraries in a nonsemihosting environment
Semihosting for AArch32 and AArch64

12.6 Reimplement the C library functions

You can create your own library functions with the same name as the Arm® Compiler for Embedded FuSa C library function you are using.

To build applications without the Arm standard C library, you must provide an alternative library that reimplements the ISO standard C library functions that your application might need, such as `printf()`. Your reimplemented library must be compliant with the Arm Embedded Application Binary Interface (AEABI).

To instruct `armclang` to not use the Arm standard C library, you must use the `armclang` options `-nostdlib` and `-nostdlibinc`. You must also use the `armlink` option `--no_scanlib` if you invoke the linker separately.

You must also use the `armclang` option `-fno-builtin` to ensure that the compiler does not perform any transformations of built-in functions. Without `-fno-builtin`, `armclang` might recognize calls to certain standard C library functions, such as `printf()`, and replace them with calls to more efficient alternatives in specific cases.


Note

If the linker sees a definition of `main()`, it automatically creates a reference to a startup symbol called `__main`. The Arm standard C library defines `__main` to provide startup code. If you use your own library instead of the Arm standard C library, then you must provide your implementation of `__main` or change the startup symbol by using the linker `--startup` option.

Example

This example reimplements the `printf()` function to simply return 1 or 0.

```
//my_lib.c:
int printf(const char *c, ...)
{
    if(!c)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Use `armclang` and `armar` to create a library from your reimplemented `printf()` function:

```
armclang --target=arm-arm-none-eabi -c -O2 -march=armv7-a -mfpu=none mylib.c -o
 mylib.o
armar --create mylib.a mylib.o
```

An example application source file `foo.c` contains:

```
//foo.c:
extern int printf(const char *c, ...);

void foo(void)
{
    printf("Hello, world!\n");
}
```

Use `armclang` to build the example application source file using the `-nostdlib`, `-nostdlibinc`, and `-fno-builtin` options. Then use `armlink` to link the example reimplemented library using the `--no_scanlib` option.

```
armclang --target=arm-arm-none-eabi -c -O2 -march=armv7-a -mfpu=none -nostdlib -
nostdlibinc -fno-builtin foo.c -o foo.o
armlink foo.o mylib.a -o image.axf --no_scanlib
```

If you do not use the `-fno-builtin` option, then the compiler transforms the `printf()` function to the `puts()` function, and the linker generates an error because it cannot find the `puts()` function in the reimplemented library.

```
armclang --target=arm-arm-none-eabi -c -O2 -march=armv7-a -mfpu=none -nostdlib -
nostdlibinc foo.c -o foo.o
armlink foo.o mylib.a -o image.axf --no_scanlib

Error: L6218E: Undefined symbol puts (referred from foo.o).
```

Related information

C library structure on page 241
--startup (armlink)

Run-time ABI for the Arm Architecture
C Library ABI for the Arm Architecture

12.7 Tailoring the image memory map to your target hardware

You can use a scatter file to define a memory map, giving you control over the placement of data and code in memory.

In your final embedded system, without semihosting functionality, you are unlikely to use the default memory map. Your target hardware usually has several memory devices located at different address ranges. To make the best use of these devices, you must have separate views of memory at load and run-time.

Scatter-loading enables you to describe the load and run-time memory locations of code and data in a textual description file known as a scatter file. This file is passed to the linker on the command line using the `--scatter` option. For example:

```
armlink --scatter scatter.scat file1.o file2.o
```

Scatter-loading defines two types of memory regions:

- Load regions containing application code and data at reset and load-time.
- Execution regions containing code and data when the application is executing. One or more execution regions are created from each load region during application startup.

A single code or data section can only be placed in a single execution region. It cannot be split.

During startup, the C library initialization code in `__main` carries out the necessary copying of code and data and the zeroing of data to move from the image load view to the execute view.


Note

The overall layout of the memory maps of devices based around the Arm®v6-M and Armv7-M architectures are fixed. This fixed layout makes it easier to port software between different systems based on these architectures.

Related information

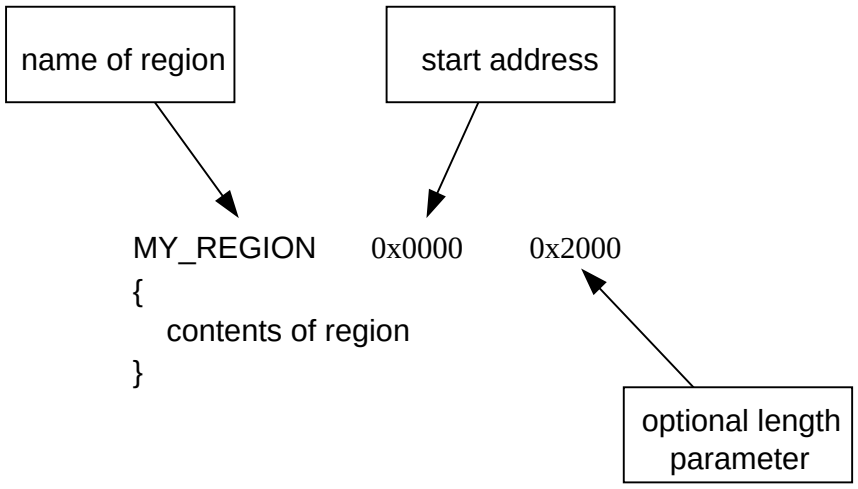
Information about scatter files
--scatter=filename (armlink)
Armv7-M Architecture Reference Manual
Armv6-M Architecture Reference Manual
Semihosting for AArch32 and AArch64

12.8 About the scatter-loading description syntax

In a scatter file, each region is defined by a header tag that contains, as a minimum, a name for the region and a start address. Optionally, you can add a maximum length and various attributes.

The scatter-loading description syntax shown in the following figure reflects the functionality provided by scatter-loading:

Figure 12-6: Scatter-loading description syntax



The contents of the region depend on the type of region:

- Load regions must contain at least one execution region. In practice, there are usually several execution regions for each load region.
- Execution regions must contain at least one code or data section, unless a region is declared with the `EMPTY` attribute. Non-`EMPTY` regions usually contain object or library code. You can use the wildcard (`*`) syntax to group all sections of a given attribute not specified elsewhere in the scatter file.

Related information

Information about scatter files
Scatter-loading images with a simple memory map

12.9 Root regions

A root region is an execution region with an execution address that is the same as its load address. A scatter file must have at least one root region.

One restriction placed on scatter-loading is that the code and data responsible for creating execution regions cannot be copied to another location. As a result, the following sections must be included in a root region:

- `__main.o` and `__scatter*.o` containing the code that copies code and data
- `__dc*.o` that performs decompression
- `Region$$Table` section containing the addresses of the code and data to be copied or decompressed.

Because these sections are defined as read-only, they are grouped by the `* (+RO)` wildcard syntax. As a result, if `* (+RO)` is specified in a non-root region, these sections must be explicitly declared in a root region using `InRoot$$Sections`.


Note

All eXecute In Place (XIP) code must be stored in root regions.

Related information

Region Table format on page 251
About placing Arm C and C++ library code

12.10 Region Table format

The Region Table is a linker-generated data structure that the Arm C library Default Initialization Sequence uses to copy, decompress, or zero-initialize code and data from its load address to its execution address. The Region Table is called `Region$$Table`.


Note

The Region Table is tightly integrated with the Arm C library Default Initialization Sequence described in Application startup. Arm reserves the right to change the format of the Region Table in future releases. Arm does not offer support on how the Arm C library uses the information in the Region Table.

The Region Table is delimited by the linker-defined symbols `Region$$Table$$Base` and `Region$$Table$$Limit`. You must place the `Region$$Table` in a root execution region. See Placement of Arm C and C++ library code for details. Each table entry comprises four 32-bit words for AArch32 ELF files and four 64-bit words for AArch64 ELF files:

Offset from start of region table entry (bytes)	Item
+0	`Load Address of source`
+4	`Execution Address of destination`
+8	`Execution Size of destination`
+12	`Address of handler routine`

The addresses are in one of three formats depending on the contents of the bottom two bits of the word:

bit 1	bit 0	Format
0	0	Absolute address
0	1	Offsets from the base of the Region Table (ROPI)
1	0	Offsets from the static base register (RWPI)

The majority of Region Table entries are absolute.

The Arm C library has different handler routines that have the following function prototype:

```
void <handler_routine>(uintptr_t <load_address>, uintptr_t <exec_address>, size_t
<exec_size>);
```

The Default Initialization Sequence processes the text entries in order, calling the `<handler_routine>` with the right parameters. In the case where the table entries are not absolute addresses, the linker adds additional veneer routines to translate the offsets into absolute addresses at runtime.

The handler routines defined by the C library are:

`<handler_routine>`	Description
`__scatterload_null`	Does nothing.
`__scatterload_copy`	Copies the number of bytes specified by `Execution Size of destination` from `Load Address of source` to `Execution Address of destination`.
`__scatterload_zeroinit`	Zero initalizes the number of bytes specified by `Execution Size of destination` starting from `Execution Address of destination`.
`__decompress`	Decompresses data starting at `Load Address of source` to `Execution Address of destination`. The size of the decompressed data is `Execution Size of Bytes`.

Examples

Using the image generated by the example described in Writing an overlay manager for manually placed overlays, the following examples show the `fromelf` output:

- To view the disassembly:

```
fromelf -disassemble image.axf
```

```
...
||Region$$Table$$Base||
        DCD      0x24002ec8
        DCD      0x00010000
        DCD      0x00000010
        DCD      0x2400003c
        DCD      0x24002ed8
        DCD      0x00010010
        DCD      0x00000244
        DCD      0x24000058
||Region$$Table$$Limit||
...
```

- To view the symbols:

```
fromelf -st image.axf

** Section #18 '.symtab' (SHT_SYMTAB)
    Size   : 13104 bytes (alignment 4)
    String table #19 '.strtab'
    Last local symbol no. 540

    Symbol table .symtab (818 symbols, 540 local)

      #  Symbol Name                 Value      Bind  Sec  Type  Vis  Size
    ============================================================================
...
    809  Region$$Table$$Base         0x24002d4c   Gb    1   --   Hi
    810  Region$$Table$$Limit        0x24002d6c   Gb    1   --   Hi
...
        EXPORT ||Region$$Table$$Base||
        EXPORT ||Region$$Table$$Limit||
...
** Section #19 '.strtab' (SHT_STRTAB)
    Size   : 8920 bytes

           #  Offset String
        ===================================
...
          451   8751: Region$$Table$$Base
          452   8771: Region$$Table$$Limit
...
```

Related information

Application startup on page 244

12.11 Placing the stack and heap

The scatter-loading mechanism provides a method for specifying the placement of the stack and heap in your image.

The application stack and heap are set up during C library initialization. You can tailor stack and heap placement by using the specially named `ARM_LIB_HEAP`, `ARM_LIB_STACK`, or `ARM_LIB_STACKHEAP` execution regions. Alternatively, if you are not using a scatter file, you can reimplement the `__user_setup_stackheap()` function.

Related information

Run-time memory models on page 254

Tailoring the C library to a new execution environment
Specifying stack and heap using the scatter file

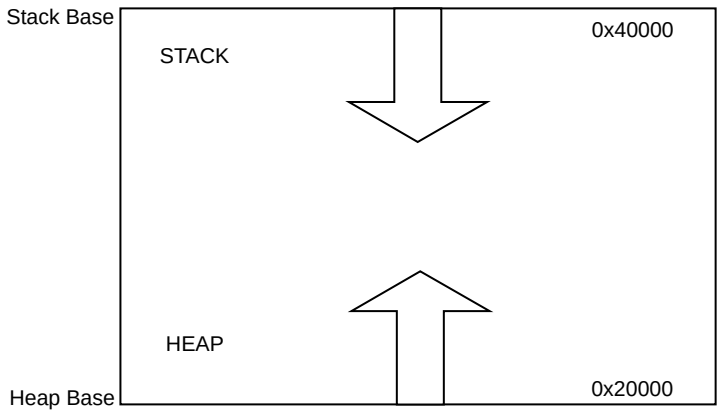
12.12 Run-time memory models

Arm® Compiler for Embedded FuSa toolchain provides one- and two-region run-time memory models.

One-region model

The application stack and heap grow towards each other in the same region of memory, see the following figure. In this run-time memory model, the heap is checked against the value of the stack pointer when new heap space is allocated. For example, when `malloc()` is called.

Figure 12-7: One-region model



One-region model routine

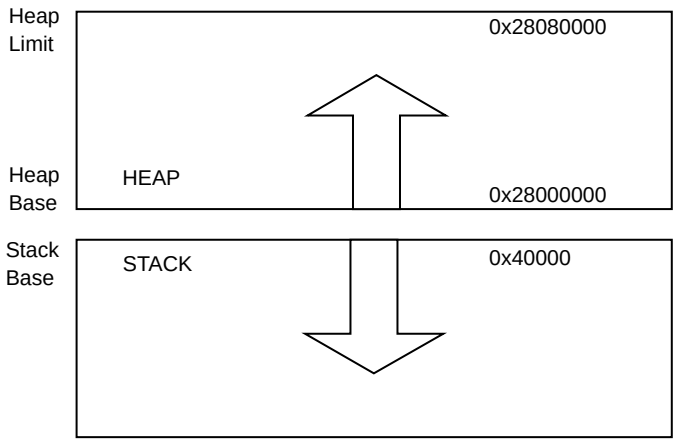
```
LOAD_FLASH ...
{
    ...
    ARM_LIB_STACKHEAP 0x20000 EMPTY 0x20000  ; Heap and stack growing towards
    { }                                      ; each other in the same region
    ...
}
```

Two-region model

The stack and heap are placed in separate regions of memory, see the following figure. For example, you might have a small block of fast RAM that you want to reserve for stack use only. For a two-region model, you must import `__use_two_region_memory`.

In this run-time memory model, the heap is checked against the heap limit when new heap space is allocated.

Figure 12-8: Two-region model



Two-region model routine

```
LOAD_FLASH ...
{
    ...
    ARM_LIB_STACK 0x40000 EMPTY -0x20000  ; Stack region growing down
    { }                                   ;
    ARM_LIB_HEAP 0x28000000 EMPTY 0x80000 ; Heap region growing up
    { }
    ...
}
```

In both run-time memory models, the stack grows unchecked.

Related information

Stack pointer initialization and heap bounds

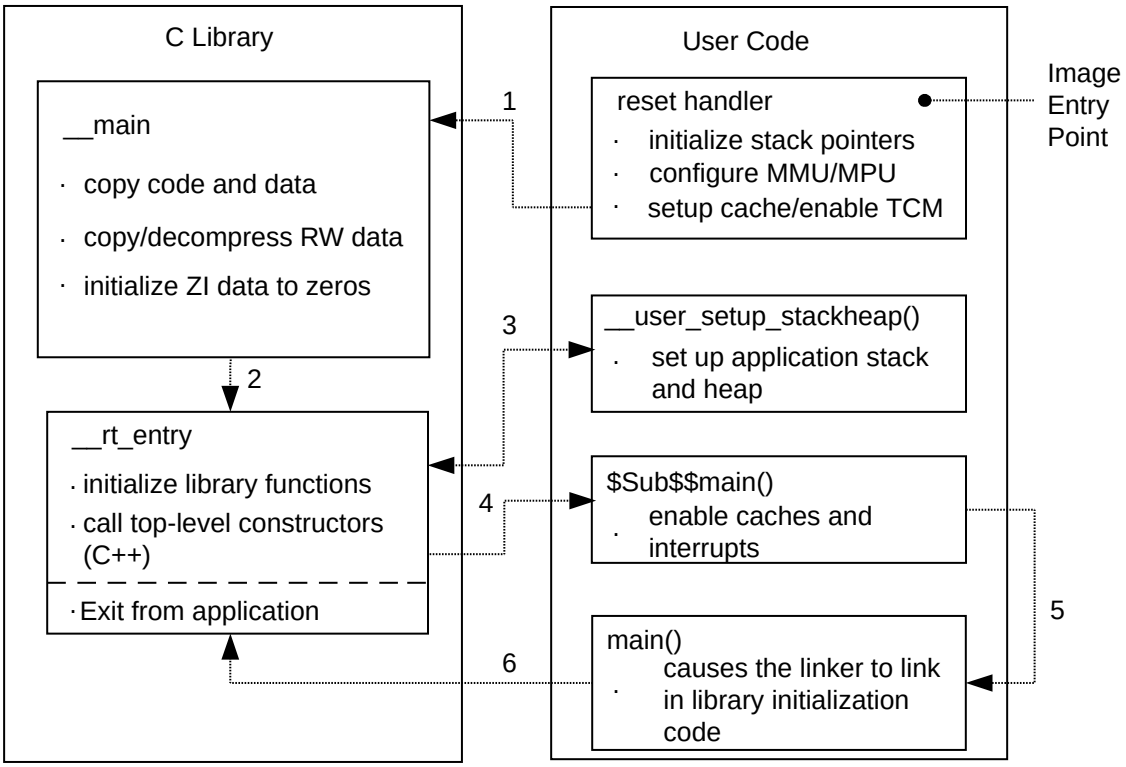
12.13 Reset and initialization

The entry point to the C library initialization routine is `__main`. However, an embedded application on your target hardware performs some system-level initialization at startup.

Embedded system initialization sequence

The following figure shows a possible initialization sequence for an embedded system based on an Arm® architecture:

Figure 12-9: Initialization sequence



If you use a scatter file to tailor stack and heap placement, the linker includes a version of the library heap and stack setup code using the linker defined symbols, `ARM_LIB_*`, for these region names. Alternatively you can create your own implementation.

The reset handler is normally a short module coded in assembler that executes immediately on system startup. As a minimum, your reset handler initializes stack pointers for the modes that your application is running in. For processors with local memory systems, such as caches, TCMs, MMUs, and MPUs, some configuration must be done at this stage in the initialization process. After executing, the reset handler typically branches to `__main` to begin the C library initialization sequence.

There are some components of system initialization, for example, the enabling of interrupts, that are generally performed after the C library initialization code has finished executing. The block of code labeled `$Sub$$main()` performs these tasks immediately before the main application begins executing.

Related information

About using $Super$$ and $Sub$$ to patch symbol definitions
Specifying stack and heap using the scatter file

12.14 The vector table

All Arm systems have a vector table. It does not form part of the initialization sequence, but it must be present for an exception to be serviced.

It must be placed at a specific address, usually `0x0`. To do this you can use the scatter-loading `+FIRST` directive, as shown in the following example.

Placing the vector table at a specific address

```
ROM_LOAD 0x0000 0x4000
{
  ROM_EXEC 0x0000 0x4000      ; root region
  {
    vectors.o (Vect, +FIRST)  ; Vector table
    * (InRoot$$Sections)      ; All library sections that must be in a
                              ; root region, for example, __main.o,
                              ; __scatter*.o, __dc*.o, and * Region$$Table
  }
  RAM 0x10000 0x8000
  {
    * (+RO, +RW, +ZI)         ; all other sections
  }
}
```

The vector table for the microcontroller profiles is very different to most Arm® architectures.

Related information

Vector table for AArch32 A and R profiles on page 257
Vector table for M-profile architectures on page 258
Information about scatter files
Scatter-loading images with a simple memory map

12.14.1 Vector table for AArch32 A and R profiles

The vector table for Arm®v7-A, Armv8-A, Armv9-A, Armv7-R, and Armv8-R profiles in AArch32 state consists of branch or load PC instructions to the relevant handlers.

If required, you can include the FIQ handler at the end of the vector table to ensure it is handled as efficiently as possible. See the following example. Using a literal pool means that addresses can easily be modified later if necessary.

Typical vector table using a literal pool

GNU assembler syntax vector table:

```
//-----------------------------------------------------------------
// Exception Vector Table
//-----------------------------------------------------------------
// Note: LDR PC instructions are used here, though branch (B) instructions
// could also be used, unless the exception handlers are >32MB away.

Vectors:
    ldr pc, Reset_Addr
```

```
    ldr pc, Undefined_Addr
    ldr pc, SVC_Addr
    ldr pc, Prefetch_Addr
    ldr pc, Abort_Addr
    b .                                 // Reserved vector
    ldr pc, IRQ_Addr
    ldr pc, FIQ_Addr


    .balign 4
Reset_Addr:     .word Reset_Handler
Undefined_Addr: .word Undefined_Handler
SVC_Addr:       .word SVC_Handler
Prefetch_Addr:  .word Prefetch_Handler
Abort_Addr:     .word Abort_Handler
IRQ_Addr:       .word IRQ_Handler
FIQ_Addr:       .word FIQ_Handler
```

Legacy `armasm` syntax vector table:

```
                AREA vectors, CODE, READONLY
                ENTRY
Vector_Table
                LDR pc, Reset_Addr
                LDR pc, Undefined_Addr
                LDR pc, SVC_Addr
                LDR pc, Prefetch_Addr
                LDR pc, Abort_Addr
                NOP                     ;Reserved vector
                LDR pc, IRQ_Addr
FIQ_Handler
                ; FIQ handler code - max 4kB in size
Reset_Addr      DCD Reset_Handler
Undefined_Addr  DCD Undefined_Handler
SVC_Addr        DCD SVC_Handler
Prefetch_Addr   DCD Prefetch_Handler
Abort_Addr      DCD Abort_Handler
IRQ_Addr        DCD IRQ_Handler
                ...
                END
```

This example assumes that you have ROM at location `0x0` on reset. Alternatively, you can use the scatter-loading mechanism to define the load and execution address of the vector table. In that case, the C library copies the vector table for you.

12.14.2 Vector table for M-profile architectures

The vector table for the microcontroller profiles consists of addresses to the relevant handlers.

The handler for exception number `<n>` is held at (`<vectorbaseaddress>` + 4 * `<n>`).

In Arm®v7-M and Armv8-M processors, you can specify the `<vectorbaseaddress>` in the Vector Table Offset Register (VTOR) to relocate the vector table. The default location on reset is `0x0` (CODE space). For Armv6-M, the vector table base address is fixed at `0x0`. The word at `<vectorbaseaddress>` holds the reset value of the main stack pointer.

Note

The least significant bit, bit[0], of each address in the vector table must be set or a HardFault exception is generated. If the table contains T32 symbol names, the Arm Compiler for Embedded FuSa toolchain sets these bits for you.

12.14.3 Vector Table Offset Register

In Arm®v7-M and Armv8-M, the Vector Table Offset Register locates the vector table in CODE, RAM, or SRAM space.

When setting a different location, the offset, in bytes, must be aligned to:

- A power of 2.
- A minimum of 128 bytes.
- A minimum of `4*<N>`, where `<N>` is the number of exceptions supported.

The minimum alignment is 128 bytes, which allows for 32 exceptions. 16 registers are reserved for system exceptions. Therefore, you can use up to 16 interrupts.

To use more interrupts, you must adjust the alignment by rounding up to the next power of two. For example, if you require 21 interrupts, then the total number of exceptions is 37, that is 21 plus 16 reserved system exceptions. The alignment must be on a 64-word boundary because the next power of 2 after 37 is 64.

Note

Implementations might restrict where the vector table can be located. For example, in Cortex®-M3 r0p0 to r2p0, the vector table cannot be in RAM space.

12.15 ROM and RAM remapping

You must consider what sort of memory your system has at address `0x0`, the address of the first instruction executed.

Note

This information does not apply to Arm®v6-M, Armv7-M, and Armv8-M profiles.

Note

This information assumes that an Arm processor begins fetching instructions at `0x0`. This is the standard behavior for systems based on Arm processors. However, some Arm processors, for example the processors based on the Armv7-A architecture, can be configured to begin fetching instructions from `0xFFFF0000`.

There has to be a valid instruction at `0x0` at startup, so you must have nonvolatile memory located at `0x0` at the moment of power-on reset. One way to achieve this is to have ROM located at `0x0`. However, there are some drawbacks to this configuration.

Example ROM/RAM remapping

This example shows a solution implementing ROM/RAM remapping after reset. The constants shown are specific to the Versatile board, but the same method is applicable to any platform that implements remapping in a similar way. Scatter files must describe the memory map after remapping.

```
;  System memory locations
Versatile_ctl_reg     EQU 0x101E0000 ; Address of control register
DEVCHIP_Remap_bit     EQU 0x100      ; Bit 8 is remap bit of control register
        ENTRY
; Code execution starts here on reset
; On reset, an alias of ROM is at 0x0, so jump to 'real' ROM.
        LDR     pc, =Instruct_2
Instruct_2
; Remap by setting remap bit of the control register
; Clear the DEVCHIP_Remap_bit by writing 1 to bit 8 of the control register
        LDR     R1, =Versatile_ctl_reg
        LDR     R0, [R1]
        ORR     R0, R0, #DEVCHIP_Remap_bit
        STR     R0, [R1]
; RAM is now at 0x0.
; The exception vectors must be copied from ROM to RAM
; The copying is done later by the C library code inside __main
; Reset_Handler follows on from here
```

12.16 About Run-Time Type Information

Run-Time Type Information (RTTI) is required when the type of a C++ class must be determined at runtime.

Arm® Compiler for Embedded FuSa 6 implements the Itanium C++ ABI and includes:

- A compiler (armclang) that can be used to compile programs written in C++.
- Two C++ libraries:
  - The C++ standard library (libc++).
  - The C++ run-time library (libc++abi).

RTTI is used by the following parts of C++:

- Exception handling.
- `dynamic_cast`.

- `typeid`.

More information about when RTTI is referenced and generated is described in section *2.9 Run-Time Type Information (RTTI)* of the Itanium C++ ABI.

RTTI for basic types such as `int` and `bool` is stored in the runtime library. Therefore, object files generated from a C++ program might reference RTTI defined in libc++abi. See section *2.9.2 Place of Emission* of the Itanium C++ ABI for more information.

The compiler also generates RTTI for a program that contains classes and structures with virtual functions.

Use of RTTI requires linking with a significant portion of libc++abi because it contains several routines involved in processing RTTI. Also, there are links to C++ exceptions, or software aborts, when `typeid` does not match.

Compiling your code the `armclang` option `-fno-rtti` does not guarantee complete removal of RTTI. The standard libc++ library is compiled to use RTTI and libc++abi includes RTTI handling functions. Therefore, you must also:

- Avoid using functions in the `std::` namespace.
- Link against stub implementations of RTTI for basic types. For more information, see Avoid linking in Run-Time Type Information.

Related information

-frtti, -fno-rtti

12.17 Avoid linking in the Arm Compiler for Embedded FuSa libraries

With the exception of the built-in helper functions you can use Arm® Compiler for Embedded FuSa without Arm library functions. You can re-implement all or part of the Arm library.

Types of library function

The following table describes the types of library function, when they are included in a system, and what action you can take to use an alternative:

Table 12-4: Types of library function

Function type	Description	Use in a system	Action
Helper	A function that the compiler might call even if no standard library headers are present.	Whenever the compiler requires a helper function to translate source code.	You can re-implement most helper functions.

Function type	Description	Use in a system	Action
Initialization	Code that runs before the `main()` function is called. Initialization code performs actions such as setting up the heap, stack, and global data required by the standard library functions.	When the C or C++ program contains a `main()` function.	You re-implement the Arm initialization code.
Standard library	A library function that is part of the C standard. These functions are called explicitly from source code.	When functions from the standard library are used in source code.	You can re-implement all standard library functions.

Helper functions

The *Run-time ABI for the Arm Architecture* document standardizes a set of helper functions that all ABI-compliant Arm Compiler for Embedded FuSa toolchains must provide. The document gives the following definition of a helper function:

A helper function is one that a relocatable file might refer to, even though its source includes no standard headers, or no headers at all. A helper function usually implements some aspect of a programming language not implemented by its standard library. For example, from C, floating-point to integer conversions.

In some cases, a helper function might implement some aspect of standard library behavior not implemented by any of its interface functions. For example, from the C library, `errno`.

A helper function might also implement an operation not implemented by the underlying hardware, for example, integer division, floating-point arithmetic, or reading and writing misaligned data.

All ABI-compliant compilers can assume that these helper functions are present. Arm Compiler for Embedded FuSa provides these helper functions in the standard C run-time library, and `armclang` uses them.

Using Arm Compiler for Embedded FuSa without libraries

To use Arm Compiler for Embedded FuSa without the standard library, you must avoid using the public helper, initialization, and standard library functions. You must re-implement these functions as required.

You must write your C or C++ code in a way that avoids the standard library, and minimizes the use of the runtime library by the compiler. Compiler and linker options are available to prevent them using the library functions.

For more information, see:

- Support for building an application without the C library.
- Avoid linking in the Arm C library.
- Avoid linking in Run-Time Type Information.
- Avoid linking in the Arm C++ libraries.

Find out which Arm library functions are used

The `armlink` option `--verbose` provides a list of all the object files that are loaded from the command-line and selected from libraries.

The following types of message identify content from the Arm libraries:

- `Searching for ARM libraries in directory <path to directory containing Arm libraries>`
- `Selecting library <path to specific Arm library>`
- `Loading member <object> from <library>`
- `definition: <symbol>`

For example:

```
...
Loading System Libraries.

Searching for ARM libraries in directory <install_path>\sw\..\sw\ARMCompiler6.21\bin
\..\lib\armlib\
...
Selecting library <install_path>\sw\..\sw\ARMCompiler6.21\bin\..\lib\armlib\c_8u.l.
...
Selecting member dc.o(c_8u.l) to define __decompress_flags.
Loading member dc.o from c_8u.l.
              definition:  __decompress_flags
              definition:  __decompress_sizes
...
```

Re-implementing standard library functions

You can re-implement all standard library functions. When there is a call to a standard library function, and if that function is re-implemented, then `armclang` calls the re-implemented function.

For more information, see:

- -fno-builtin.
- Reimplement the C library functions.

Related information

--verbose
ABI for the Arm Architecture

12.17.1 Avoid linking in the Arm C library

The C runtime libraries provided with Arm® Compiler for Embedded FuSa 6 are suitable for various Arm-based projects. However, some projects might have certain requirements that mean it is necessary to avoid using all or part of the standard C library.


Note

This topic includes descriptions of [COMMUNITY] features. See Support level definitions.

For example:

- The project must use a certified Functional Safety (FuSa) C library to make it easier to fulfill the safety requirements for the project.
- The project uses alternative libraries provided by the Operating System (OS) vendor.
- The project has some custom requirements to re-implement certain C library functionality.

The following sections expand on the information provided in Standalone C library functions.

The following sections do not:

- Describe how to fully avoid the C++ library.
- Explain how to develop your startup and initialization code that must run before the `main` function is reached.

The C libraries provided in Arm Compiler for Embedded FuSa 6

Arm Compiler for Embedded FuSa 6 provides the following C runtime libraries:

- C standard library (standardlib).
- C Micro-library (microlib).

The C standardlib is the default C library that projects are likely to use. The microlib is an alternative to the standard C library. Microlib focuses in particular on smaller code size, but with some documented limitations and restrictions.

Build options required to avoid the standard C library

The following compiler and linker options are required to avoid the C library being used explicitly by your build files and implicitly by the compiler:

Compiler options

- `-fno-builtin` prevents the compiler from transforming standard C library function calls based on built-in knowledge about how those functions behave.

  This option applies to functions such as `printf`, but does not apply to `__builtin_<name>` functions, despite the name. The compiler knows something about functions such as `printf`, and sometimes transforms the source code based on that understanding.

However, the compiler still expects the library to provide an implementation of those functions.

For example, if your code calls `printf("hello, world\n")`, the compiler might convert it into `puts("hello, world")` because it knows from the descriptions of those two functions in the C standard that they perform the same operations. But the `puts()` function cannot perform all the operations of `printf` by itself. If you write a more complicated call involving formatting such as `%d`, then use this option to ensure the compiler emits a call to the `printf` library function.

- `-nobuiltininc` prevents the compiler from using the built-in header files.
- `-nostdlib` prevents the compiler from using the Arm standard C and C++ libraries.
- `-nostdlibinc` prevents the compiler from using the Arm standard C and C++ library header files.


Note

To use the Arm FuSa C library with the libc++ header files, you must use the `-nobuiltininc`, `-nostdlibinc`, and `-nostdlib` options. The FuSa C library is different from the Arm standard C library because it is designed to work without the built-in header files.

If you are working in a freestanding, non-hosted, environment you can specify the [COMMUNITY] option `-ffreestanding`. This option:

- Asserts that compilation targets a freestanding environment.
- Implies `-fno-builtin`.
- Sets the macro `STD_C_HOSTED` to 0.

Linker options

- `--no_scanlib` prevents the linker from scanning the Arm libraries to resolve references. As a consequence of using this option, the Arm supplied libraries are not used by the linker and you must include your own libraries.

Source code changes to avoid the C library

The function label `main()` has a special significance. The presence of a `main()` function forces the linker to link in the initialization code in `__main`. The `__main` function calls the following initialization functions:

- `__scatterload` (scatter-loading memory initialization code).
- `__rt_entry` (runtime library initialization code).

Without a function labeled `main()`, the initialization sequence is not linked in, and as a result, some standard C library functionality is not supported.

To prevent a reference to `__main`, either:

- Specify a different `main` function, for example, `my_main()`.
- Link with the `--no_startup` option.

Related information

Application startup on page 244
Avoid linking in Run-Time Type Information on page 267
-fno-builtin
-ffreestanding
-nostdlib
-nostdlibinc
--scanlib, --no_scanlib
--startup=symbol, --no_startup
__rt_entry

12.17.2 Avoid linking in the Arm C++ libraries

You can avoid the libc++ library by not calling functions from the standard library. However, it is more difficult to avoid the libc++abi runtime library because the compiler can implicitly refer to functions and Run-Time Type Information (RTTI) defined in that library.

That is, you do not need to include any headers or call functions directly for the compiler to emit a call to a function defined in the runtime library. The libc++abi library contains implementations of these functions and some additional low-level support for libc++. Major components include:

- RTTI
- Exceptions
- New and Delete
- Terminate
- Static initialization guards
- Pure virtual abort handler

To link with your own ABI-compliant runtime library, specify the following `armclang` command-line options:

- `-fno-exceptions` to disable the generation of code needed to support C++ exceptions.
- `-fno-rtti` to avoid `typeinfo` in object files and ensure no references to libc++abi `typeinfo` functionality.
- `-nobuiltininc` to exclude the built-in header files.
- `-nostdlib` to pass `--noscanlib` to the linker and do not perform `printf` optimization. This option disables the inclusion of both the C and C++ libraries.
- `-nostdlibinc` to not add the include and `include/libcxx` include directories.
- `-nostdinc++` to disable standard `#include` directories for the C++ standard library.
- `-I <project_runtime_library_header>` to specify the location of your ABI-compliant runtime library headers.
- `-L <project_runtime_library>` to specify the location of your ABI-compliant runtime library.

For more information on how to avoid linking in libc++abi, see Avoid linking in Run-Time Type Information.

Related information

-fexceptions, -fno-exceptions
-frtti, -fno-rtti
-I
-nobuiltininc
-nostdlib
-nostdlibinc

12.17.3 Avoid linking in Run-Time Type Information

When targeting a system with a limited amount of memory, you might want to avoid linking in Run-Time Type Information (RTTI) to reduce the overall application size.

libc++ is compiled with RTTI. You can avoid using libc++ by not calling any `std::` namespace functions. However, the typeinfos in libc++abi might still be referenced.

Avoiding libc++abi

Compiling all source code with the `armclang` option `-fno-rtti` does not guarantee complete removal of RTTI from the linked program.

However, RTTI is not used, and `armclang` does not generate calls to the RTTI handling functions in libc++abi, when all the following conditions are true:

- `-fno-exceptions` is used to disable C++ exceptions.
- `dynamic_cast` is not used in the application, or is used in such a way that RTTI is not required.
- `typeid` is not used in the application.

If your code includes `typeid`, then specifying `-fno-rtti` results in an error. However, an error is output for `dynamic_cast` only if the way it is used requires RTTI.

To ensure you avoid RTTI for the basic types being linked in from libc++abi, you must provide stub implementations of RTTI for basic types as a placeholder.

Providing such stubs is sufficient to link the application, but not to use the C++ features that depend on RTTI. That is, C++ exceptions, `dynamic_cast`, and `typeid`.

Example: Stub implementation to avoid linking with libc++abi

The following C++ example shows the use of stub implementations. The example contains an assembly file `typeinfo.s` that has a section named `unused_rtti` with stubs representing all the RTTI basic types in libc++abi.

1. Create the `hello.cpp` file containing the following code:

```
#include <iostream>
```

```
int main(void)
{
    std::cout << "Hello World!" << std::endl;

    return 0;
}
```

2. Create the `typeinfo.s` file containing the source code provided in typinfo.s example source code.
3. Create the scatter file `scatter.sct` containing the following:

```
LOAD_ROM 0x0000 0x20000
{
    EXEC_ROM 0x0000 0x20000
    {
        *(+RO)
    }

    SRAM 0x20000000 0x6000
    {
        *(+RW, +ZI)
    }

    UNUSED_RTTI +0x0 UNINIT
    {
        typeinfo.o(unused_rtti)
    }

    ARM_LIB_STACKHEAP 0x20008000 EMPTY -0x1000 {}
}
```

   The scatter file explicitly places the `unused_rtti` section in an UNINIT section to ensure that the RTTI stubs do not occupy any memory.
4. Build the C++ and assembler code with the following commands:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -c -fno-rtti -fno-exceptions
 hello.cpp -o hello.o
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -c typeinfo.s -o typeinfo.o
```

5. Link the object files as follows:

```
armlink --cpu=Cortex-M3 --scatter=scatter.sct --map --load_addr_map_info --
verbose --list=hello.lst -o hello.axf hello.o typeinfo.o
```

   The linker command includes an option to generate a listings file named `hello.lst` that includes:

   - The memory map and symbol listing for the final image.
   - The verbose output to show how the linker resolved references to definitions, including the references to the RTTI stubs.

   The memory map shows that the execution region containing the RTTI data is treated as `UNINIT` and does not occupy any space:

```
Execution Region UNUSED_RTTI (Exec base: 0x20000d18, Load base: 0x00016610, Size:
 0x0000000c, Max: 0xffffffff, ABSOLUTE, UNINIT)
```

```
Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name
    Object

0x20000d18        -       0x0000000c   Zero   RW          30    unused_rtti
    typeinfo.o
```

6. Run the following `fromelf` command:

```
fromelf --text -c -d -s -v hello.axf -o hello.txt
```

`fromelf` generates a listing file containing:

- Code and disassembly listing.
- Data section contents.
- Symbol table.
- Verbose output for section information.

Related information

About Run-Time Type Information on page 260
-fexceptions, -fno-exceptions
-frtti, -fno-rtti

12.17.4 C++ functions you can re-implement

There are some C++ functions that you can re-implement.

`__cxa_deleted_virtual`

Use when a virtual function is explicitly deleted, for example:

```
virtual void test() = delete;
```

Do not use. This code is unlikely to be common.

`__cxa_guard_acquire` and `__cxa_guard_release`

Use when a function local `static` is used, for example:

```
void test() {
    static MyClass my_c;
    ...
}
```

Instead of using function local `static`, use pointers to place new instantiated objects. Dynamically initialized memory using `new` is also possible, but it is expected that you want to avoid this. These pointers are inline versions of the guards, for example:

```
void test() {
    alignas(MyClass) static char buf[sizeof(MyClass)];
    static MyClass *cp = nullptr;
    if (cp == nullptr) {
```

```
        cp = new(buf) MyClass;
    }
}
```

However, this method can compromise concurrency in a multi-threaded system. The `__cxa_guard_acquire` system is designed to cope with multiple instances of the function running concurrently on different cores, and still arranges that the variable is initialized once only. In that situation, this version of the C++ code with a static pointer could suffer a race condition in which the `MyClass` constructor runs twice.

If your program is entirely single-threaded then using the static pointer is not an issue.

See also *-fthreadsafe-statics*, *-fno-threadsafe-statics <https://developer.arm.com/documentation/109443/6-22-2LTS/armclang-Reference/armclang-Command-line-Options/-fthreadsafe-statics-fno-threadsafe-statics>*.

`__cxa_pure_virtual`

When a pure virtual function is used, for example:

```
virtual void test() = 0;
```

Do not use. Provide a default implementation that aborts if called. This method is not ideal because the toolchain can provide diagnostics if it can prove a pure virtual function is going to be called, and if there is no implementation of a pure virtual function.

`new` and `delete`

Use when non-placement `new` is called and there is no user-defined global `operator::new` overload or type specific `new` overload present.


Note

The implementation in libc++abi is only for those applications that are not using libc++.

Provide your own implementations of `new` and `delete`. The libc++ header contains the prototypes. Because these functions are not qualified, it is likely that you have to take the prototypes from the standard so that they can be qualified. Arm expects that most bare-metal applications use placement `new` and `delete`. Therefore, such applications can avoid dynamic memory allocation.

12.18 Local memory setup considerations

Many Arm processors have on-chip memory management systems, such as Memory Management Units (MMU) or Memory Protection Units (MPU). These devices are normally set up and enabled during system startup.

Therefore, the initialization sequence of processors with local memory systems requires special consideration.

The C library initialization code in `__main` is responsible for setting up the execution time memory map of the image. Therefore, the run-time memory view of the processor must be set up before branching to `__main`. This means that any MMU or MPU must be set up and enabled in the reset handler.

Tightly Coupled Memories (TCM) must also be enabled before branching to `__main`, normally before MMU/MPU setup, because you generally want to scatter-load code and data into TCMs. You must be careful that you do not have to access memory that is masked by the TCMs when they are enabled.

You might also encounter problems with cache coherency if caches are enabled before branching to `__main`. Code in `__main` copies code regions from their load address to their execution address, essentially treating instructions as data. As a result, some instructions can be cached in the data cache, in which case they are not visible to the instruction path.

To avoid these coherency problems, enable caches after the C library initialization sequence finishes executing.

Related information

Cortex-A Series Programmer's Guide for Armv8-A
Cortex-A Series Programmer's Guide for Armv7-A
Cortex-R Series Programmer's Guide for Armv7-R

12.19 Stack pointer initialization

As a minimum, your reset handler must assign initial values to the stack pointers of any execution modes that are used by your application.

Example stack pointer initialization

In this example, the stacks are located at `stack_base`:

```
; *************************************************************
; This example does not apply to M-profile
; *************************************************************
Len_FIQ_Stack    EQU     256
Len_IRQ_Stack    EQU     256
stack_base       DCD     0x18000
;
Reset_Handler
    ; stack_base could be defined above, or located in a scatter file
    LDR     R0, stack_base ;
    ; Enter each mode in turn and set up the stack pointer
    MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit    ; Interrupts disabled
    MOV     sp, R0
    SUB     R0, R0, #Len_FIQ_Stack
    MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit    ; Interrupts disabled
    MOV     sp, R0
    SUB     R0, R0, #Len_IRQ_Stack
    MSR     CPSR_c, #Mode_SVC:OR:I_Bit:OR:F_Bit    ; Interrupts disabled
    MOV     sp, R0
    ; Leave processor in SVC mode
```

The `stack_base` symbol can be a hard-coded address, or it can be defined in a separate assembler source file and located by a scatter file.

The example allocates 256 bytes of stack for Fast Interrupt Request (FIQ) and Interrupt Request (IRQ) mode, but you can do the same for any other execution mode. To set up the stack pointers, enter each mode with interrupts disabled, and assign the appropriate value to the stack pointer.

The stack pointer value set up in the reset handler is automatically passed as a parameter to `__user_initial_stackheap()` by C library initialization code. Therefore, this value must not be modified by `__user_initial_stackheap()`.

Related information

Specifying stack and heap using the scatter file
Cortex-M3 Embedded Software Development

12.20 Hardware initialization

In general, it is beneficial to separate all system initialization code from the main application. However, some components of system initialization, for example, enabling of caches and interrupts, must occur after executing C library initialization code.

Use of $Sub and $Super

You can make use of the `$Sub` and `$Super` function wrapper symbols to insert a routine that is executed immediately before entering the main application. This mechanism enables you to extend functions without altering the source code.

This example shows how `$Sub` and `$Super` can be used in this way:

```
extern void $Super$$main(void);

void $Sub$$main(void)
{
    cache_enable();     // enables caches
    int_enable();       // enables interrupts
    $Super$$main();     // calls original main()
}
```

The linker replaces the function call to `main()` with a call to `$Sub$$main()`. From there you can call a routine that enables caches and another to enable interrupts.

The code branches to the real `main()` by calling `$Super$$main()`.

Related information

Use of $Super$$ and $Sub$$ to patch symbol definitions

12.21 Execution mode considerations

You must consider the mode in which the main application is to run. Your choice affects how you implement system initialization.


Note

This does not apply to Arm®v6-M, Armv7-M, and Armv8-M profiles.

Much of the functionality that you are likely to implement at startup, both in the reset handler and `$Sub$$main`, can only be done while executing in privileged modes, for example, on-chip memory manipulation, and enabling interrupts.

If you want to run your application in a privileged mode, this is not an issue. Ensure that you change to the appropriate mode before exiting your reset handler.

If you want to run your application in User mode, however, you can only change to User mode after completing the necessary tasks in a privileged mode. The most likely place to do this is in `$Sub$$main()`.


Note

The C library initialization code must use the same stack as the application. If you need to use a non-User mode in `$Sub$$main` and User mode in the application, you must exit your reset handler in System mode, which uses the User mode stack pointer.

12.22 Target hardware and the memory map

It is better to keep all information about the memory map of a target, including the location of target hardware peripherals and the stack and heap limits, in your scatter file, rather than hard-coded in source or header files.

Mapping to a peripheral register

Conventionally, addresses of peripheral registers are hard-coded in project source or header files. You can also declare structures that map on to peripheral registers, and place these structures in the scatter file.

For example, if a target has a timer peripheral with two memory mapped 32-bit registers, a C structure that maps to these registers is:

```
struct
{
    volatile unsigned ctrl;           /* timer control */
    volatile unsigned tmr;            /* timer value   */
} timer_regs;
```

Note

You can also use `__attribute__((section(".ARM.__at_<address>")))` to specify the absolute address of a variable.

Placing the mapped structure

To place this structure at a specific address in the memory map, you can create an execution region containing the module that defines the structure. The following example shows an execution region called `TIMER` that locates the `timer_regs` structure at `0x40000000`:

```
ROM_LOAD 0x24000000 0x04000000
{
; ...
    TIMER 0x40000000 UNINIT
    {
        timer_regs.o (+ZI)
    }
    ; ...
}
```

It is important that the contents of these registers are not zero-initialized during application startup, because this is likely to change the state of your system. Marking an execution region with the `UNINIT` attribute prevents ZI data in that region from being zero-initialized by `__main`.

Related information

Placement of functions and data at specific addresses on page 190
__attribute__((section("name"))) variable attribute

12.23 Execute-only memory

Execute-Only Memory (XOM) allows only instruction fetches. Read and write accesses are not allowed.

Execute-only memory allows you to protect your intellectual property by preventing executable code being read by users. For example, you can place firmware in execute-only memory and load user code and drivers separately. Placing the firmware in execute-only memory prevents users from trivially reading the code.

Note

The Arm architecture does not directly support execute-only memory. Execute-only memory is supported at the memory device level.

Related information

Building applications for execute-only memory on page 274

12.24 Building applications for execute-only memory

Placing code in execute-only memory prevents users from trivially reading that code.

About this task


Note

Link-Time Optimization (LTO) does not honor the `armclang` option `-mexecute-only` option. If you use the `armclang` options `-flto` or `-Omax`, then the compiler cannot generate execute-only code.

Procedure

1. Compile your C or C++ code using the `-mexecute-only` option.

   ```
   armclang --target=arm-arm-none-eabi -march=armv7-m -mexecute-only -c test.c -o
    test.o
   ```

   The `-mexecute-only` option prevents the compiler from generating any data accesses to the code sections.

   To keep code and data in separate sections, the compiler disables the placement of literal pools inline with code.

   Compiled execute-only code sections in the ELF object file are marked with the `SHF_ARM_NOREAD` flag.
2. Specify the memory map to the linker using either of the following:
   - The `+XO` selector in a scatter file.
   - The `armlink --xo-base` option on the command-line.

     ```
     armlink --xo-base=0x8000 test.o -o test.axf
     ```

   The XO execution region is placed in a separate load region from the RO, RW, and ZI execution regions.


Note

If you do not specify `--xo-base`, then by default:

- The XO execution region is placed immediately before the RO execution region, at address `0x8000`.
- All execution regions are in the same load region.

Related information

Execute-only memory on page 274
Compiling with -mexecute-only generates an empty .text section on page 276
-mexecute-only (armclang)
--execute_only (armasm)

--xo_base=address (armlink)
AREA directive

12.25 Compiling with -mexecute-only generates an empty .text section

Compiling with `-mexecute-only` always generates an empty `.text` section that is read-only. That is, a section that does not have the `SHF_ARM_PURECODE` attribute.

The linker normally removes the empty `.text` section during unused section elimination. However, the unused section elimination does not occur when:

- The image has no entry point.
- You specify one of the following linker options:
  - `--no_remove`
  - `--keep (<object-file-name>(.text))`

If you use a scatter file to merge eXecute-Only (XO) and Read-Only (RO) sections into a single executable region, then the XO sections lose the XO attribute and become RO.

When compiling with `-fno-function-sections`, all functions are placed in the `.text` section with the `SHF_ARM_PURECODE` attribute. As a result, there are two sections with the name `.text`, one with and one without the `SHF_ARM_PURECODE` attribute. You cannot select between the two `.text` sections by name. Therefore, you must use attributes as the selectors in the scatter file to differentiate between XO and RO sections.

Examples

The following example shows how Arm® Compiler for Embedded FuSa 6.22.2 handles `.text` sections:

1. Create the file `example.c` containing:

```
void foo() {}

int main() {
  foo();
}
```

2. Compile the program and examine the object file with `fromelf`.

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-M3 -mexecute-only -c -o
 example.o example.c
fromelf example.o
```

The output shows that `Section #2` is the empty RO `.text` section:

```
...
```

```
** Section #1 '.strtab' (SHT_STRTAB)
 Size : 148 bytes


** Section #2 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
 Size : 0 bytes (alignment 4)
 Address: 0x00000000


** Section #3 '.text.foo' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR +
 SHF_ARM_PURECODE]
 Size : 2 bytes (alignment 4)
 Address: 0x00000000


** Section #4 '.ARM.exidx.text.foo' (SHT_ARM_EXIDX) [SHF_ALLOC + SHF_LINK_ORDER]
 Size : 8 bytes (alignment 4)
 Address: 0x00000000
 Link to section #3 '.text.foo'


** Section #5 '.rel.ARM.exidx.text.foo' (SHT_REL)
 Size : 8 bytes (alignment 4)
 Symbol table #13 '.symtab'
 1 relocations applied to section #4 '.ARM.exidx.text.foo'


** Section #6 '.text.main' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR +
 SHF_ARM_PURECODE]
 Size : 10 bytes (alignment 4)
 Address: 0x00000000
...
```

3. Create the file `example.scat` containing:

```
LR_XO 0x10000

{
  ER_MAIN_FOO 0x10000
  {
     example.o(.text*)
  }
}

LR_2 0x20000
{
  ER_REST 0x20000
  {
     *(+RO, +ZI)
  }
  ARM_LIB_STACKHEAP 0x80000 EMPTY -0x1000 {}
}
```

4. Create an image file with `armlink` and examine the image file with `fromelf`:

```
armlink --scatter example.scat -o example_scat.axf example.o
fromelf example_scat.axf
```

The output shows that `Section #1` has the `SHF_ARM_PURECODE` attribute:

```
...

** Section #1 'ER_MAIN_FOO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR +
 SHF_ARM_PURECODE]
 Size : 16 bytes (alignment 4)
```

```
 Address: 0x00010000


** Section #2 'ER_REST' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
 Size : 604 bytes (alignment 4)
 Address: 0x00020000
...
```

5. Repeat the link again with the linker option `--no_remove` and examine the image file with `fromelf`.

```
armlink --scatter example.scat --no_remove -o example_scat.axf example.o
fromelf example_scat.axf
```

The output shows that `Section #1` does not have the `SHF_ARM_PURECODE` attribute:

```
...

** Section #1 'ER_MAIN_FOO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 16 bytes (alignment 4)
    Address: 0x00010000


** Section #2 'ER_REST' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 604 bytes (alignment 4)
    Address: 0x00020000
...
```

The empty RO `.text` section is no longer removed and is placed in the same execution region as `.text.main` and `.text.foo`. Therefore, these sections become read-only.

The same result is obtained when linking with `--keep example.o(.text)` or if there is no `main` or no entry point.

6. To ensure that the sections remain as execute-only, either:
   - Change the scatter file to use the XO attribute selector as follows:

```
LR_XO 0x10000

{
  ER_MAIN_FOO 0x10000
  {
     example.o(+XO)
  }
}

LR_2 0x20000
{
  ER_REST 0x20000
  {
     *(+RO, +ZI)
  }
  ARM_LIB_STACKHEAP 0x80000 EMPTY -0x1000 {}
}
```

- Explicitly place sections in their execution regions. However, compiling with `-fno-function sections` generates two `.text` sections with different attributes:

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-M3 -mexecute-only -fno-
function-sections -c -o example.o example.c

fromelf example.o
...
** Section #1 '.strtab' (SHT_STRTAB)
    Size   : 107 bytes


** Section #2 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 0 bytes (alignment 4)
    Address: 0x00000000


** Section #3 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR +
 SHF_ARM_PURECODE]
    Size   : 14 bytes (alignment 4)
    Address: 0x00000000
...
```

  In this case, differentiating the sections by name only is not possible. If unused section elimination does not remove the empty `.text` sections, the attribute selectors are required to place the sections in different output sections.

12.26 Integer division by zero errors in C and C++ code

Integer division by zero in C and C++ code is undefined behavior, and the compiler does not guarantee a specific behavior for such code.

Integer division by zero behavior for processors that support hardware division instructions

For processors that support hardware division instructions, the behavior depends on the Divide by Zero support of the processor:

- Trapping Divide by Zero errors.
- Returning a zero result on Divide by Zero.

For more information about the Divide by Zero support, see the Technical Reference Manual (TRM) for your processor.

Integer division by zero behavior for processors that do not support hardware division instructions

For processors that do not support hardware division instructions, such as the `SDIV` and `UDIV` instructions, you cannot rely on the C and C++ library helper function `__aeabi_idiv0()` to trap and identify integer division by zero errors. Instead, you must manually test the denominator before the division operation takes place. For example:

```
#include <signal.h>
```

```
int divide(const int numerator, const int denominator)
{
    if (denominator == 0)
    {
        return raise(SIGFPE);
    }
    else
    {
        return numerator / denominator;
    }
}
```

Note

You can trap integer division by zero at run-time with the Undefined Behavior Sanitizer (UBSan) functionality. See Overview of Undefined Behavior Sanitizer for more information.

12.27 Floating-point division by zero errors in C and C++ code

The floating-point division by zero behavior that results from assumptions made by `armclang` might be undesirable.

AArch64 state behavior

The Floating-point Control Register (FPCR) and Floating-point Status Register (FPSR) are AArch64 registers. For AArch64 state, setting the Divide by Zero floating-point exception trap enable bit `FPCR.DZE` to `1` tells the processor that a floating-point Divide by Zero operation causes a synchronous exception within the processor instead of updating the Divide by Zero cumulative floating-point exception bit `FPSR.DZC`. The exception handler routine can then decide whether to set the `FPSR.DZC` to `1` to indicate that a Divide by Zero operation occurred.

Note

If floating-point exception trapping is not supported by the Arm®v8-A implementation, then the processor ignores any attempt to set `FPCR.DZE` to `1`.

`armclang` assumes that the `FPCR.DZE` bit is never set to `1`. `armclang` also incorrectly assumes that a processor always automatically sets `FPSR.DZC` to `1` to indicate that a divide-by-zero operation has occurred. Therefore, `armclang` can move a comparison with `0.0f` after a potential divide-by-zero operation, because it assumes a divide-by-zero operation does not affect program flow. However, if the implementation supports floating-point exception trapping and your code sets `FPCR.DZE` to `1`, a divide-by-zero operation does affect the program flow and might cause a processor exception. If the processor does not support floating-point exception trapping, then setting `FPCR.DZE` to `1` might result in unexpected runtime behavior. Therefore, make sure your code is written such that `armclang` avoids placing the division before the comparison.

AArch32 state behavior

For AArch32, both fields `DZE` and `DZC` are in the combined Floating-point Status and Control Register (FPSCR). For AArch32 state, `armclang` makes the same assumption as in AArch64 state, that a divide-by-zero operation does not affect program flow.

Example: Common code pattern to guard against division by zero

A common code pattern is to guard against division by zero, as shown in the following C code example:

```
float func(float x, float y)
{
    if (y != 0.0f)
    {
        return x/y;
    }
    return x;
}
```

However, because of the assumptions `armclang` makes about floating-point instructions, it might compile the example C code for AArch64 state as follows:

```
    fdiv s2, s0, s1
    fcmp s1, #0.0
    fcsel s0, s2, s0, ne
    ret
```

This example shows that the division is performed before the comparison, and executed unconditionally, which might be undesirable.

The following examples show how to work around the division by zero behavior in source code.

Example: Work around by declaring the divisor as volatile

By declaring the divisor as `volatile`, `armclang` expects that the value of `y` might change between reads. `volatile` forces `armclang` to produce more conservative code, where the comparison necessarily comes before the division:

```
float func(float x, volatile float y)
{
    if (y != 0.0f)
    {
        return x/y;
    }
    return x;
}
```

Example: Work around by using inline assembly

An alternative solution is to perform the division operation using an inline assembly block. Declaring the inline assembly block as `volatile` prevents `armclang` from optimizing that block. For example, for AArch64 state:

```
float func(float x, float y)
```

```
{
    float ret;
    if (y != 0.0f)
    {
        __asm volatile ("fdiv %s0, %s1, %s2"
          :"=w"(ret)
          :"w"(x), "w"(y)
          :);
    } else {
        ret = x;
    }
    return ret;
}
```

12.28 Dealing with leftover debug data for code and data removed by armlink

`armlink` eliminates unused functions to reduce code size. However, because the debug information is not embedded on a function level but at the object level, the linker is unable to remove the associated unused debug information.

When `armlink` removes code, it resolves references to addresses in the removed range to `0x0` by default. Therefore, any debug information for that code now points to address `0x00000000`. Resolving to `0x00000000` is a problem when the target processor has a vector table at that address and you want to set a breakpoint at that address. Therefore, use `--dangling-debug-address` to specify an unused address to use to resolve references to the removed code.

Note

You could temporarily turn off the automatic removal of unused code with `--no-remove`. However, this option increases the overall code size.

The default `armclang` option is `-ffunction-sections`. Therefore, when compiling a translation unit containing two functions, the resulting `.o` file contains a separate code section for each function. However, the debug data sections contain data for both functions.

At link time, one of the code sections might be referenced but the other is not. Therefore, if the linker wants to retain debug data for only one function, the `.o` file contains sections that have debug data for both functions. When the linker applies all the address relocations to the debug data relative to the retained function, then it generates an acceptable image. However, there remain all the address relocations for debug data relative to the function that is absent. In this case, the linker applies the relocations for these data relative to the address supplied by `--dangling-debug-address`.

Typically, you use a high address well away from your code, but not at the very top of the address range, for example:

```
armlink --dangling_debug_address 0xF0000000
```

This command forces any leftover debug data to be moved well away from the startup code around `0x0` that you are trying to debug.

You must have enough virtual address space after the address specified with `--dangling-debug-address` so that all the debug data relocated to that region safely points to nothing.

Related information

-ffunction-sections, -fno-function-sections
--dangling-debug-address=address
--remove, --no_remove

12.29 Building images that are compatible with third-party tools

Embedded applications require explicit control over the grouping and placement of output image components. Arm tools are able to understand the image placement. However, fundemental differences exist when third-party tools load images produced with an incompatible component structure.

Arm® Compiler for Embedded FuSa provides scatter-loading features that can support complex memory maps, such as overlapping regions or placing code and data into non-consecutive areas of memory. Not all tools can handle the complex layouts that Arm Compiler for Embedded FuSa supports. Therefore, Arm Compiler for Embedded FuSa provides a simplified mode when the following properties of the regions are ensured:

- Each load region has a single relocation.
- There is at least one RO region and one root region.
- None of the regions are overlays or overlap.

Arm Compiler for Embedded FuSa provides the following `armlink` command-line options to modify the output symbols and the addresses of the output image:

- `--elf-output-format` to modify the symbols and addresses of the output image to be compatible with third-party tools.
- `--scatterload-enabled` or `--no-scatterload-enabled` to enable or disable the generation of scatter-loading.

Region table generation is disabled when the `--no-scatterload-enabled` option is used, or when the `--elf-output-format` is set to `gnu`. As such, the linker does not generate region table related symbols such as `Load$$LR`. Applications that make use of `Load$$LR` fail to link.

Using the `__attribute__((section(".ARM.__at_<address>")))` variable attribute also allows third-party tools to load the load region. However, this attribute might not work fully because the load region misses the RO section.

Related information

--elf-output-format
--scatterload-enabled, --no-scatterload-enabled
__attribute__((section("name"))) variable attribute
Scatter-loading Features

13. Security features supported in Arm Compiler for Embedded FuSa

A security-related feature either detects security flaws in your source code or adds protection through a combination of code generation and library code. A feature mitigates against a potential security threat, such as Return Oriented Programming (ROP) or Jump Oriented Programming (JOP).

Arm® Compiler for Embedded FuSa supports the following security features:

- Armv8-M Security Extension (CMSE).
- Stack protection.
- Branch target protection.
- Return address signing.
- Return address signing hardening.
- Stack memory tagging.
- Heap memory tagging.
- Automatic variable initialization.
- Control Flow Integrity (CFI) sanitizer.
- Undefined Behavior Sanitizer (UBSan).
- Straight-Line Speculation (SLS) hardening.

Note

Varying the stack location at program startup to increase address diversity of the stack pointer is also good practice to reduce the risk of attacks.

For information on protecting memory, see Memory-safety best practices.

Armv8-M Security Extension

You access CMSE using the `armclang` option `-mcmse` which enables the code generation for the Secure state.

Threat Model

The attacker is trying to access secrets stored on the system or call into code that must not be accessible to a user.

Assumptions

- The attacker has compromised Non-secure state and can perform any action permitted in Non-secure state.
- You have designed and implemented your system according to the best practices described in Armv8-M Secure software guidelines for Armv8-M Secure software guidelines.

- The Secure state is not compromised.

Protection mechanism

CMSE is not a simple compiler feature that can protect arbitrary code. You must architect your system with CMSE in mind.

CMSE provides support as follows:

- Hardware that supports CMSE has a Secure and Non-secure state, where Secure state is mostly not visible from Non-secure state.
- A gateway region that is accessible to Non-secure state provides entry points from Non-secure to Secure state.

You must build Secure state code and Non-secure state code as two separate programs. Arm Compiler for Embedded FuSa provides support for:

- Code generation for Non-secure entry functions.
- Code generation for calling Non-secure functions.
- Intrinsics to query memory permissions.
- Linker generation of gateway veneers.

Your Secure state code must perform the following operations to ensure the Secure state is not compromised:

- Sanitize and verify the addresses provided by the Non-secure state.
- Clear all state, such as floating-point registers, before returning to the Non-secure state.

For more information, see Overview of building Secure and Non-secure images with the Armv8-M Security Extension.

Stack protection

You access stack protection using the set of `armclang` options `-fstack-protector*` to make code generation changes that detect stack smashing attacks.

Threat Model

The attacker is trying to perform a ROP attack by overwriting the return address on the stack using an overflow.

Assumptions

Stack protection assumes the attacker:

- Has no access to higher level privilege.
- Does not have control of the stack.
- Only has read-only access to code.
- Can provide input to the program.
- Can disassemble code.
- Does not know the value of `__stack_chk_guard` or the location of `__stack_chk_guard`.

- Can make as many attempts as they like to attack the program.

Protection mechanism

- You write code to initialize the value used as a canary value at a known location `__stack_chk_guard`. We recommend using a different value every time the program starts.
- The compiler inserts a canary value into the stack frame such that a buffer overrun that would overwrite the return address would have to overwrite the canary value.
- On function exit, the compiler adds a check on the canary value to see if it matches the value in `__stack_chk_guard`. If the check fails, calls a user-defined function that usually terminates the program.

For more information, see Overview of memory tagging.

Branch target protection

You access branch target protection using the `armclang` option `-mbranch-protection` to make code generation changes for Armv8.5-A and later or Armv8.1-M targets that support the PACBTI extension to prevent uncontrolled branches.

Library support for branch protection is available in the `*a.*` variants. See C and C++ library naming conventions.

Threat Model

The attacker is trying to perform a ROP or JOP attack, by overwriting an address of an indirect jump.

Assumptions

Branch target protection assumes the attacker:

- Has no access to higher level privilege.
- Only has read-only access to code.
- Has control of the stack, because other protections have not been applied or have failed.
- Can disassemble code.
- Can make as many attempts as they like to attack the program.

Protection mechanism

- You enable branch protection for the system on Armv8.5-A and later or Armv8.1-M or for the memory pages covering the program in AArch64.
- An indirect branch that does not land on a landing pad instruction causes an abort. This restricts the set of places that an attacker that compromises the system can jump to.
- The compiler inserts landing pad instructions that can be jumped to.
- The assembler author is responsible for adding landing pad instructions.

For more information, see -mbranch-protection.

Return address signing

You access return address signing using the `armclang` option `-mbranch-protection`. This option makes code generation changes to protect the return address for Armv8.3-A and later and Armv8.1-M targets that support the PACBTI extension.

Library support for return address signing is available in the `*a.*` variants. See C and C++ library naming conventions.

Threat Model

Return address signing assumes the attacker:

- Has no access to higher level privilege.
- Only has read-only access to code.
- Can provide input to the program.
- Can disassemble code.
- Can make as many attempts as they like to attack the program.

Protection mechanism

Return address signing is similar to stack protection, but instead of a canary value, the return address on the stack is signed on function entry and authenticated on function exit. An attacker must be able to replace the return address with a signed value that successfully authenticates.

For more information, see Armv8.1-M PACBTI extension mitigations against ROP and JOP style attacks.

Return address signing hardening

You access return address signing hardening using the `armclang` option `-mharden-pac-ret` with `-mbranch-protection=pac-ret`. Together, these options make code generation changes that harden the return address signing.

Threat Model

The PACMAN: Attacking ARM Pointer Authentication with Speculative Execution paper describes a high level threat model.

Protection mechanism

The PACMAN method uses malicious software to brute-force Pointer Authentication Codes (PAC). To protect against this technique, you must enable return address signing using `-mbranch-protection=pac-ret`, and use `-mharden-pac-ret` to harden the return address signing. When you use these options:

- The PAC is authenticated.
- The XPAC family of instructions strips the PAC.
- The compiler performs a load of the return address after the PAC has been stripped.


Note

Return address signing hardening mitigation is incompatible with execute only. Code that contains the return address signing hardening mitigation must have read permission.

For more information, see -mharden-pac-ret.

Stack memory tagging

You access stack memory tagging using the `armclang` option `-fsanitize=memtag-stack`. This option makes code generation changes for Armv8.5-A and later targets that support the Memory Tagging Extension (MTE) to protect against stack smashing attacks.

Threat Model

The Arm_Memory_Tagging_Extension_Whitepaper.pdf describes a high level threat model.

Protection mechanism

- The compiler generates a pseudo-random initial tag value when allocating a stack frame.
- The compiler aligns stack objects on the stack to match the tag granularity.
- The compiler allocates stack slots using a tag value derived from the initial tag. The intention is that adjacent allocations get a different tag.
- The hardware can detect a tag mismatch and cause an abort if it is configured to do so.

There are a finite number of tags so this mechanism provides probablistic protection only. The immediate + offset form is not subject to checks.

For more information, see Overview of memory tagging.

Heap memory tagging

You access heap memory tagging using the `armclang` option `-fsanitize=memtag-heap` and the `armclang` symbol `__use_memtag_heap`. `-fsanitize=memtag-heap` makes code generation changes for Armv8.5-A and later targets that support the Memory Tagging Extension (MTE) to protect against heap overflow attacks. `__use_memtag_heap` controls linker library selection.

Threat Model

The Arm_Memory_Tagging_Extension_Whitepaper.pdf describes a high level threat model.

Protection mechanism

- The `malloc`, `free`, `calloc`, and `realloc` functions are modified to set and clear tags for allocations.
- Assign tags to adjacent allocations.
- Change tags on `free`.
- If hardware is configured to do so, the hardware can detect a tag mismatch and cause an abort.

For more information, see Overview of memory tagging.

Automatic variable initialization

You access automatic variable initialization using the `armclang` option `-ftrivial-auto-var-init` to initialize automatic variables with either a pattern or zeroes, or set them to uninitialized.

For more information, see -ftrivial-auto-var-init.

Control Flow Integrity sanitizer

You access the CFI sanitizer using the `armclang` option `-fsanitize=cfi` to implement a number of CFI schemes. These schemes are designed to abort the program on detection of certain forms of undefined behavior that can potentially allow attackers to subvert the control flow of the program.

CFI requires that you also enable Link-Time Optimization (LTO) with the `armclang` option `-flto` and the `armlink` option `--lto`.

For more information, see Overview of Control Flow Integrity.

Undefined Behavior Sanitizer

You access UBSan using the `armclang` option `-fsanitize=<ubsan_check>` to instruct the compiler to insert code instrumentation to catch undefined behaviors during runtime.

Threat Model

Code with undefined behavior is a target for hackers.

Protection mechanism

The compiler inserts runtime checks for common instances of undefined behavior such as integer overflow and Divide by Zero. Although the runtime checks cost a single digit amount of performance and add to code size, they are low enough to deploy in production. The supported UBSan modes give you control over how to handle an undefined behavior.


Note

The `armclang` option `-fsanitize=function` is incompatible with execute only. Code compiled with this option must have read permission.

For more information, see Overview of Undefined Behavior Sanitizer.

Straight-Line Speculation hardening

A processor might speculatively execute the instructions immediately following a change in control flow, including:

- Exception generating instructions (`SVC`, `HVC`, `SMC`, `UNDEF`, `BRK`).
- Exception returns (`ERET`).
- Unconditional direct branches (`B`, `BL`).
- Unconditional indirect branches (`BR`, `BLR`).
- Function returns (`RET`).

The `armclang` option `-mharden-sls` generates code that helps prevent a processor from speculating past affected indirect branch instructions on AArch64 targets. For information about other branch instructions, see the Straight-line speculation whitepaper.

For more information, see Overview of Straight-Line Speculation hardening.

13.1 How optimization can interfere with security

You have applied the relevant security features or secure coding guidelines to your programs using the supported Arm® Compiler for Embedded FuSa security features. However, that work can be undone by some Arm Compiler for Embedded FuSa optimizations and leave your programs vulnerable.

We recommend using lower optimization levels for files with secure code. If you use higher optimization levels, then you can use the following mitigation:

- Removal of code that seems redundant to the compiler, but is an important check for some security property. For example:
  - Elimination of unused sections can remove a function or variable that is critical to security. To prevent the removal of a function or variable, you can mark that function or variable in source code with the `__attribute__((used))` attribute. Alternatively, you can use the `armlink` option `--keep=<section_id>`.
  - Inlining can affect whether a function is protected. To prevent a function being inlined, specify the `__attribute__((noinline))` function attribute or the `armclang` option `-fno-inline-functions`.
- Removal of memory stores that seem to be redundant to the compiler because the variable is not used afterwards, but leaves sensitive data in memory. For example, removal of a seemingly unused variable can prevent a function from being protected. To prevent the removal of a variable that is essential to a security feature, declare that variable as `volatile` or use the `__attribute__((used))` attribute.
- Changes in code that do not allow the same time execution paths, therefore allowing side channel attacks.

The following online resources describe some of the relevant issues:

- CWE-733: Compiler Optimization Removal or Modification of Security-critical Code.
- Insecure Compiler Optimization.
- Insecure Compiler Optimization: Pointer Arithmetic.
- Security flaws caused by compiler optimizations.
- The Security Implications Of Compiler Optimizations On Cryptography - A Review.

Related information

Hardware errata and vulnerabilities on page 292
Effect of the volatile keyword on compiler optimization on page 73
-fno-inline-functions

__attribute__((used)) function attribute
-keep=section_id (armlink)
Elimination of unused sections

13.2 Hardware errata and vulnerabilities

Hardware errata are bugs in the Arm hardware design or implementation. Arm publishes Errata Notice to document errata and their mitigations.

Arm Security Center

The Arm Security Center contains information on security-related resources such as vulnerabilities and errata that have a security advisory.

How to find the SDEN for your hardware

Get the published Software Developers Errata Notice (SDEN) for your hardware:

1. Browse to Arm Developer Documentation.
2. Enter **Software Developers Errata Notice** in the search field.
3. Select the **Software Developers Errata Notice** document type.
4. Locate the SDEN for your hardware. Expand **All Categories > IP Products > Processors** and locate the processor for the image you are building.


Note

All SDENs are published as PDFs.

Finding vulnerability KBAs and Product Advisory Notices

KnowledgeBase articles (KBAs) and Product Advisory Notices (PANs) describe vulnerabilities. Vulnerabilities are identified by IDs with the format `CVE-<yyyy>-<xxxxx>`, where `<yyyy>` is the year the vulnerability is disclosed. To find KBAs and PANs:

1. Browse to Arm Developer Documentation.
2. Enter **CVE** in the search field.

How to apply software mitigations for your Arm hardware

The SDEN for your hardware provides a summary of the published errata in the *Release Information* section, and the ID of each erratum. The *Revisions Affected* indicates which hardware revisions the errata affects. A detailed description of each erratum is provided in the appropriate *Category* section, and details of any known mitigations.

Where errata mitigations are available that can be applied using Arm® Compiler for Embedded FuSa, the mitigations are provided through either `armclang` mitigations or `armlink` patches:

- To apply `armclang` mitigations, use the `-mfix-<feature>-<ID>` option. `<feature>` might be the name of a processor, or the name of an Arm Compiler for Embedded FuSa feature. `<ID>` can be one of the following combinations:
  - `<name>-<erratum_ID>`, for example `aes-1742098`
  - `<erratum_ID>`, for example `835769`.
  - `<vulnerability_ID>`, for example `cve-2021-42574`.

  For example:
  - To apply the AES erratum fix 1742098 for the Cortex®-A57 processor, use the command-line option `-mfix-cortex-a57-aes-1742098`.
  - To apply the fix for the CMSE vulnerability cve-2021-42574, use the command-line option `-mfix-cmse-cve-2021-42574`.


Note

Some mitigations might be automatically applied for affected targets. The mitigation description indicates whether you need to use the `-mfix*` option or the alternate `-mno-fix*` option.

- To apply `armlink` patches, use the `--branchpatch=<processor>-<erratum_ID>` option.

  For example, to apply erratum 835769 for the Cortex-A53 processor, use the command-line option `--branchpatch=cortex-a53-835769`.

  To get information on the modification made to the program by the workaround, specify the `--info=patches` option.

13.3 Overview of building Secure and Non-secure images with the Armv8-M Security Extension

Arm® Compiler for Embedded FuSa tools allow you to build images that run in the Secure state of the Armv8-M Security Extension. You can also create an import library package that developers of Non-secure images must have for those images to call the Secure image.


Note

- The Armv8-M Security Extension is not supported when building Read-Only Position Independent (ROPI) and Read/Write Position Independent (RWPI) images.
- We recommend that Secure world software adds the value `0xfef5eda5` to the top of the main and process stacks. Adding this value is known as stack sealing. CMSIS 5.8.0 or later handles stack sealing. For more information, see CMSIS 5.

  For more information about stack sealing, see the advisory notice Armv8-M Stack Sealing vulnerability.

To build an image that runs in the Secure state you must include the `<arm_cmse.h>` header in your code, and compile using the `armclang` command-line option `-mcmse`. Compiling in this way makes the following features available:

- The Test Target, `TT`, instruction.
- `TT` instruction intrinsics.
- Non-secure function pointer intrinsics.
- The `__attribute__((cmse_nonsecure_call))` and `__attribute__((cmse_nonsecure_entry))` function attributes.

On startup, your Secure code must set up the Security Attribution Unit (SAU) and call the Non-secure startup code.

Important considerations when compiling Secure and Non-secure code

Be aware of the following when compiling Secure and Non-secure code:

- Mixing objects compiled for Armv8-M.baseline and Armv8-M.mainline could potentially leak sensitive data, because Armv8-M.baseline does not support the Floating-Point Extension. Therefore, the compiler cannot generate code to clear the Secure floating-point registers when performing a Non-secure call. If any object is compiled for the Armv8-M.mainline architecture, all files containing CMSE attributes must be compiled for the Armv8-M.mainline architecture.
- You can compile your Secure and Non-secure code in C or C++, but the boundary between the two must have C function call linkage.
- You cannot pass C++ objects, such as classes and references, across the security boundary.
- You must not throw C++ exceptions across the security boundary.
- The value of the predefined macro `__ARM_FEATURE_CMSE` indicates what Armv8-M Security Extension features are supported.
- Compile Secure code with the maximum capabilities for the target. For example, if you compile with no FPU then the Secure functions do not clear floating-point registers when returning from functions declared as `__attribute__((cmse_nonsecure_entry))`. Therefore, the functions could potentially leak sensitive data.
- Structs with undefined bits caused by padding and half-precision floating-point members are currently unsupported as arguments and return values for Secure functions. Using such structs might leak sensitive information. Structs that are large enough to be passed by reference are also unsupported and produce an error.
- The following cases are not supported when compiling with the `armclang` option `-mcmse` and produce an error:
  - Variadic entry functions.
  - Entry functions with arguments that do not fit in registers, because there are either many arguments or the arguments have large values.
  - Non-secure function calls with arguments that do not fit in registers, because there are either many arguments or the arguments have large values.

- You might have more arguments in entry functions or Non-secure function calls than can fit in registers. In this situation, you can pass a pointer to a struct containing all the arguments. For example:

```
typedef struct {
    int p1;
    int p2;
    int p3;
    int p4;
    int p5;
} Params;

void your_api(int p1, int p2, int p3, int p4, int p5) {
    Params p1 = { p1, p2, p3, p4, p5 };
    your_api_implementation(&p1);
}
```

  Here, `your_api_implementation(&p1)` is the call to your existing function, with fewer than the maximum number of 4 arguments allowed.

How a Non-secure image calls a Secure image using veneers

Calling a Secure image from a Non-secure image requires a transition from Non-secure to Secure state. A transition is initiated through Secure gateway veneers. Secure gateway veneers decouple the addresses from the rest of the Secure code.

An entry point in the Secure image, `<entryname>`, is identified with:

```
__acle_se_entryname:

entryname:
```

The calling sequence is as follows:

1. The Non-secure image uses the branch `BL` instruction to call the Secure gateway veneer for the required entry function in the Secure image:

```
    bl    entryname
```

2. The Secure gateway veneer consists of the `SG` instruction and a call to the entry function in the Secure image using the `B` instruction:

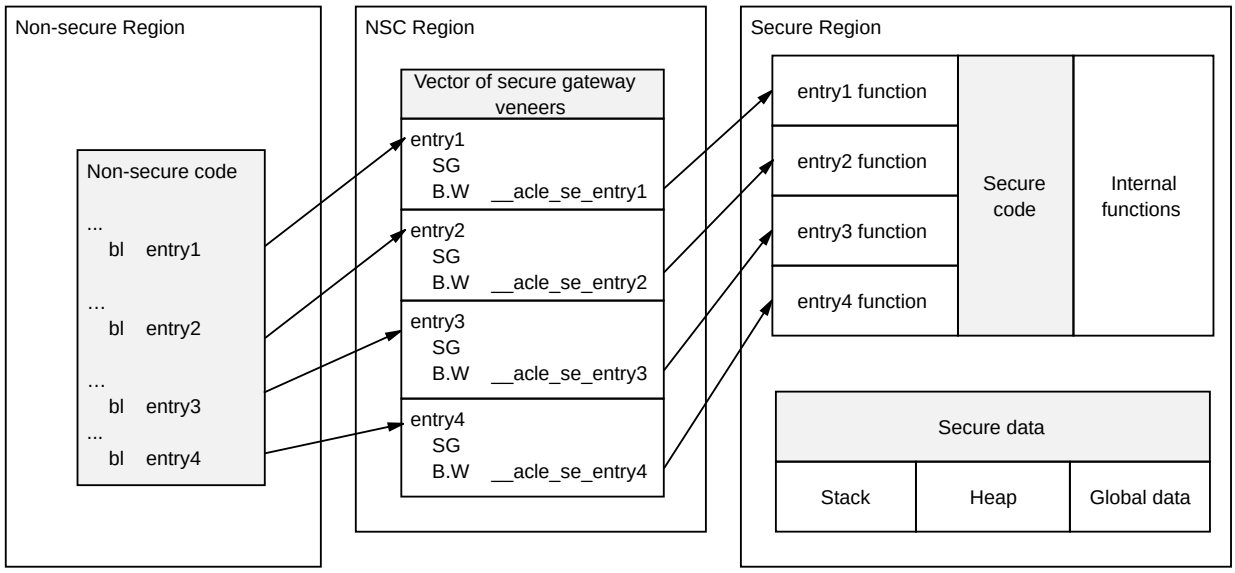
```
entryname

    SG
    B.W        __acle_se_entryname
```

3. The Secure image returns from the entry function using the `BXNS` instruction:

```
    bxns  lr
```

The following figure is a graphical representation of the calling sequence, but for clarity, the return from the entry function is not shown:



Import library package

An import library package identifies the entry functions available in a Secure image. The import library package contains:

- An interface header file, for example `myinterface.h`. You manually create this file using any text editor.
- An import library, for example `importlib.o`. `armlink` generates this library during the link stage for a Secure image.

Note

You must do separate compile and link stages:

- To create an import library when building a Secure image.
- To use an import library when building a Non-secure image.

Related information

Building a Secure image using the Armv8-M Security Extension on page 297
Building a Secure image using a previously generated import library on page 302
Building a Non-secure image that can call a Secure image on page 301
Whitepaper - Armv8-M Architecture Technical Overview
-mcmse
__attribute__((cmse_nonsecure_call)) function attribute
__attribute__((cmse_nonsecure_entry)) function attribute
Predefined macros
TT instruction intrinsics
Non-secure function pointer intrinsics
B instruction
BL instruction

BXNS instruction
SG instruction
TT, TTT, TTA, TTAT instruction
Placement of CMSE veneer sections for a Secure image

13.4 Building a Secure image using the Armv8-M Security Extension

When building a Secure image you must also generate an import library that specifies the entry points to the Secure image. The import library is used when building a Non-secure image that needs to call the Secure image.

Before you begin

The following procedure is not a complete example, and assumes that your code sets up the Security Attribution Unit (SAU) and calls the Non-secure startup code.


Note

We recommend that Secure world software adds the value `0xfef5eda5` to the top of the main and process stacks. Adding this value is known as stack sealing. CMSIS 5.8.0 handles stack sealing. See CMSIS 5 for more information. For more information about stack sealing, see the advisory notice Armv8-M Stack Sealing vulnerability

Procedure

1. Create an interface header file, `myinterface_v1.h`, to specify the C linkage for use by Non-secure code:

```
#ifdef __cplusplus
extern "C" {
#endif

int entry1(int x);
int entry2(int x);

#ifdef __cplusplus
}
#endif
```

2. In the C program for your Secure code, `secure.c`, include the following:

```
#include <arm_cmse.h>
#include "myinterface_v1.h"

int func1(int x) { return x; }
int __attribute__((cmse_nonsecure_entry)) entry1(int x) { return func1(x);  }
int __attribute__((cmse_nonsecure_entry)) entry2(int x) { return entry1(x); }

int main(void) { return 0; }
```

   In addition to the implementation of the two entry functions, the code defines the function `func1()` that is called only by Secure code.


Note

If you are compiling the Secure code as C++, then you must add `extern "C"` to the functions declared as `__attribute__((cmse_nonsecure_entry))`.

3. Create an object file using the `armclang` command-line option `-mcmse`:

```
$ armclang -c --target=arm-arm-none-eabi -march=armv8-m.main -mcmse secure.c -o
 secure.o
```

4. Enter the following command to see the disassembly of the machine code that `armclang` generates:

```
$ armclang -c --target=arm-arm-none-eabi -march=armv8-m.main -mcmse -S secure.c
```

The disassembly is stored in the file `secure.s`, for example:

```
    .text
    ...
    .code 16
    .thumb_func
    ...
func1:
    .fnstart
    ...
    bx lr
    ...
__acle_se_entry1:
entry1:
    .fnstart
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl func1
    ...
    pop.w {r7, lr}
    ...
    bxns lr
    ...
__acle_se_entry2:
entry2:
    .fnstart
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl entry1
    ...
    pop.w {r7, lr}
    bxns lr
    ...
main:
    .fnstart
    ...
    movs r0, #0
    ...
    bx lr
    ...
```

An entry function does not start with a Secure Gateway (SG) instruction. The two symbols `__acle_se_<entry_name>` and `<entry_name>` indicate the start of an entry function to the linker.

5. Create a scatter file containing the `Veneer$$CMSE` selector to place the entry function veneers in a Non-Secure Callable (NSC) memory region.

```
LOAD_REGION 0x0 0x3000
{
    EXEC_R 0x0
    {
        *(+RO,+RW,+ZI)
    }
    EXEC_NSCR 0x4000 0x1000
    {
        *(Veneer$$CMSE)
    }
    ARM_LIB_STACK 0x700000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
...
```

6. Link the object file using the `armlink` command-line option `--import-cmse-lib-out` and the scatter file to create the Secure image:

```
$ armlink secure.o -o secure.axf --cpu 8-M.Main --import-cmse-lib-out
 importlib_v1.o --scatter secure.scf
```

In addition to the final image, the link in this example also produces the import library, `importlib_v1.o`, for use when building a Non-secure image. Assuming that the section with veneers is placed at address `0x4000`, the import library consists of a relocatable file containing only a symbol table with the following entries:

Symbol type	Name	Address
`STB_GLOBAL, SHN_ABS, STT_FUNC`	`entry1`	`0x4001`
`STB_GLOBAL, SHN_ABS, STT_FUNC`	`entry2`	`0x4009`

When you link the relocatable file corresponding to this assembly code into an image, the linker creates veneers in a section containing only entry veneers.


Note

If you have an import library from a previous build of the Secure image, you can ensure that the addresses in the output import library do not change when producing a new version of the Secure image. To ensure that the addresses do not change, specify the `--import-cmse-lib-in` command-line option together with the `--import-cmse-lib-out` option. However, make sure the input and output libraries have different names.

7. Enter the following command to see the entry veneers that the linker generates:

```
$ fromelf --text -s -c secure.axf
```

The following entry veneers are generated in the EXEC_NSCR eXecute-Only (XO) region for this example:

```
...
```

```
** Section #3 'EXEC_NSCR' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR +
 SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
        0x00004000:    e97fe97f    ....    SG        ; [0x3e08]
        0x00004004:    f7fcb85e    ..^.    B         __acle_se_entry1 ; 0xc4
    entry2
        0x00004008:    e97fe97f    ....    SG        ; [0x3e10]
        0x0000400c:    f7fcb86c    ..l.    B         __acle_se_entry2 ; 0xe8
...
```

The section with the veneers is aligned on a 32-byte boundary and padded to a 32-byte boundary.

If you do not use a scatter file, the entry veneers are placed in an `ER_XO` section as the first execution region, for example:

```
...
** Section #1 'ER_XO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00008000

    $t
    entry1
        0x00008000:    e97fe97f    ....    SG        ; [0x7e08]
        0x00008004:    f000b85a    ..Z.    B.W       __acle_se_entry1 ; 0x80bc
    entry2
        0x00008008:    e97fe97f    ....    SG        ; [0x7e10]
        0x0000800c:    f000b868    ..h.    B.W       __acle_se_entry2 ; 0x80e0
...
```

Next steps

After you have built your Secure image:

1. Pre-load the Secure image onto your device.
2. Deliver your device with the pre-loaded image, together with the import library package, to a party who develops the Non-secure code for this device. The import library package contains:
   - The interface header file, `myinterface_v1.h`.
   - The import library, `importlib_v1.o`.

Related information

Building a Secure image using a previously generated import library on page 302
Building a Non-secure image that can call a Secure image on page 301
Whitepaper - Armv8-M Architecture Technical Overview
-c armclang option
-march armclang option
-mcmse armclang option
-S armclang option
--target armclang option
__attribute__((cmse_nonsecure_entry)) function attribute

SG instruction
--cpu armlink option
--import_cmse_lib_in armlink option
--import_cmse_lib_out armlink option
--scatter armlink option
--text fromelf option

13.5 Building a Non-secure image that can call a Secure image

If you are building a Non-secure image that is to call a Secure image, the Non-secure code must be written in C. You must also obtain the import library package that was created for that Secure image.

Before you begin

The following procedure assumes that you have the import library package that is created in Building a Secure image using the Arm®v8-M Security Extension. The package provides the C linkage that allows you to compile your Non-secure code as C or C++.

The import library package identifies the entry points for the Secure image.

Procedure

1. Include the interface header file in the C program for your Non-secure code, `nonsecure.c`, and use the entry functions as required.

```
#include <stdio.h>
#include "myinterface_v1.h"

int main(void) {
    int val1, val2, x;

    val1 = entry1(x);
    val2 = entry2(x);

    if (val1 == val2) {
        printf("val2 is equal to val1\n");
    } else {
        printf("val2 is different from val1\n");
    }

    return 0;
}
```

2. Create an object file, `nonsecure.o`.

```
$  armclang -c --target arm-arm-none-eabi -march=armv8-m.main nonsecure.c -o
 nonsecure.o
```

3. Create a scatter file for the Non-secure image, but without the Non-Secure Callable (NSC) memory region.

```
LOAD_REGION 0x8000 0x3000
{
    ER 0x8000
    {
```

```
        *(+RO,+RW,+ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
...
```

4. Link the object file using the import library, `importlib_v1.o`, and the scatter file to create the Non-secure image.

```
$  armlink nonsecure.o importlib_v1.o -o nonsecure.axf --cpu=8-M.Main --scatter
 nonsecure.scat
```

Related information

Building a Secure image using the Armv8-M Security Extension on page 297
Whitepaper - Armv8-M Architecture Technical Overview
-march armclang option
--target armclang option
--cpu armlink option
--scatter armlink option

13.6 Building a Secure image using a previously generated import library

You can build a new version of a Secure image and use the same addresses for the entry points that were present in the previous version. You specify the import library that is generated for the previous version of the Secure image and generate another import library for the new Secure image.

Before you begin

The following procedure is not a complete example, and assumes that your code sets up the Security Attribution Unit (SAU) and calls the Non-secure startup code.

The following procedure assumes that you have the import library package that is created in Building a Secure image using the Arm®v8-M Security Extension.

Procedure

1. Create an interface header file, `myinterface_v2.h`, to specify the C linkage for use by Non-secure code:

```
#ifdef __cplusplus
extern "C" {
#endif

int entry1(int x);
int entry2(int x);
int entry3(int x);
int entry4(int x);
```

```
#ifdef __cplusplus
}
#endif
```

2. Include the following in the C program for your Secure code, `secure.c`:

```
#include <arm_cmse.h>
#include "myinterface_v2.h"

int func1(int x) { return x; }
int __attribute__((cmse_nonsecure_entry)) entry1(int x) { return func1(x);  }
int __attribute__((cmse_nonsecure_entry)) entry2(int x) { return entry1(x); }
int __attribute__((cmse_nonsecure_entry)) entry3(int x) { return func1(x) +
 entry1(x);  }
int __attribute__((cmse_nonsecure_entry)) entry4(int x) { return entry1(x) *
 entry2(x); }

int main(void) { return 0; }
```

In addition to the implementation of the two entry functions, the code defines the function `func1()` that is called only by Secure code.


Note

If you are compiling the Secure code as C++, then you must add `extern "C"` to the functions declared as `__attribute__((cmse_nonsecure_entry))`.

---

3. Create an object file using the `armclang` command-line option `-mcmse`:

```
$ armclang -c --target arm-arm-none-eabi -march=armv8-m.main -mcmse secure.c -o
 secure.o
```

4. To see the disassembly of the machine code that is generated by `armclang`, enter:

```
$ armclang -c --target arm-arm-none-eabi -march=armv8-m.main -mcmse -S secure.c
```

The disassembly is stored in the file `secure.s`, for example:

```
    .text
    ...
    .code 16
    .thumb_func
    ...
func1:
    .fnstart
    ...
    bx lr
    ...
__acle_se_entry1:
entry1:
    .fnstart
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl func1
    pop.w {r7, lr}
    ...
    bxns lr
    ...
__acle_se_entry4:
entry4:
    .fnstart
    .save    {r7, lr}
```

```
    push    {r7, lr}
    ...
    bl entry1
    ...
    pop.w {r7, lr}
    bxns lr
    ...
main:
    .fnstart
    ...
    movs r0, #0
    ...
    bx lr
    ...
```

An entry function does not start with a Secure Gateway (SG) instruction. The two symbols `__acle_se_<entry_name>` and `<entry_name>` indicate the start of an entry function to the linker.

5. Create a scatter file containing the `Veneer$$CMSE` selector to place the entry function veneers in a Non-Secure Callable (NSC) memory region.

```
LOAD_REGION 0x0 0x3000
{
    EXEC_R 0x0
    {
        *(+RO,+RW,+ZI)
    }
    EXEC_NSCR 0x4000 0x1000
    {
        *(Veneer$$CMSE)
    }
    ARM_LIB_STACK 0x700000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
...
```

6. Link the object file using the `armlink` command-line options `--import-cmse-lib-out` and `--import-cmse-lib-in`, together with the preprocessed scatter file to create the Secure image:

```
$ armlink secure.o -o secure.axf --cpu 8-M.Main --import-cmse-lib-out
 importlib_v2.o --import-cmse-lib-in importlib_v1.o --scatter secure.scf
```

In addition to the final image, the link in this example also produces the import library, `importlib_v2.o`, for use when building a Non-secure image. Assuming that the section with veneers is placed at address `0x4000`, the import library consists of a relocatable file containing only a symbol table with the following entries:

Symbol type	Name	Address
STB_GLOBAL, SHN_ABS, STT_FUNC	entry1	0x4001
STB_GLOBAL, SHN_ABS, STT_FUNC	entry2	0x4009
STB_GLOBAL, SHN_ABS, STT_FUNC	entry3	0x4021
STB_GLOBAL, SHN_ABS, STT_FUNC	entry4	0x4029

When you link the relocatable file corresponding to this assembly code into an image, the linker creates veneers in a section containing only entry veneers.

7. Enter the following command to see the entry veneers that the linker generates:

```
$ fromelf --text -s -c secure.axf
```

The following entry veneers are generated in the EXEC_NSCR eXecute-Only (XO) region for this example:

```
...
** Section #3 'EXEC_NSCR' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR +
 SHF_ARM_NOREAD]
    Size   : 64 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
        0x00004000:    e97fe97f    ....    SG        ; [0x3e08]
        0x00004004:    f7fcb85e    ..^.    B         __acle_se_entry1 ; 0xc4
    entry2
        0x00004008:    e97fe97f    ....    SG        ; [0x3e10]
        0x0000400c:    f7fcb86c    ..l.    B         __acle_se_entry2 ; 0xe8

...

    entry3
        0x00004020:    e97fe97f    ....    SG        ; [0x3e28]
        0x00004024:    f7fcb872    ..r.    B         __acle_se_entry3 ; 0x10c
    entry4
        0x00004028:    e97fe97f    ....    SG        ; [0x3e30]
        0x0000402c:    f7fcb888    ....    B         __acle_se_entry4 ; 0x140

...
```

The section with the veneers is aligned on a 32-byte boundary and padded to a 32-byte boundary.

If you do not use a scatter file, the entry veneers are placed in an ER_XO section as the first execution region. The entry veneers for the existing entry points are placed in a CMSE veneer section. For example:

```
...
** Section #1 'ER_XO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00008000

    $t
    entry3
        0x00008000:    e97fe97f    ....    SG        ; [0x7e08]
        0x00008004:    f000b87e    ..~.    B.W       __acle_se_entry3 ; 0x8104
    entry4
        0x00008008:    e97fe97f    ....    SG        ; [0x7e10]
        0x0000800c:    f000b894    ....    B.W       __acle_se_entry4 ; 0x8138

...

** Section #4 'ER$$Veneer$$CMSE_AT_0x00004000' (SHT_PROGBITS) [SHF_ALLOC +
 SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
        0x00004000:    e97fe97f    ....    SG        ; [0x3e08]
        0x00004004:    f004b85a    ..Z.    B.W       __acle_se_entry1 ; 0x80bc
```

```
    entry2
        0x00004008:    e97fe97f    ....    SG          ; [0x3e10]
        0x0000400c:    f004b868    ..h.    B.W         __acle_se_entry2 ; 0x80e0

...
```

Next steps

After you have built your updated Secure image:

1. Pre-load the updated Secure image onto your device.
2. Deliver your device with the pre-loaded image, together with the new import library package, to a party who develops the Non-secure code for this device. The import library package contains:
   - The interface header file, `myinterface_v2.h`.
   - The import library, `importlib_v2.o`.

Related information

Building a Secure image using the Armv8-M Security Extension on page 297
Building a Non-secure image that can call a Secure image on page 301
Whitepaper - Armv8-M Architecture Technical Overview
-c armclang option
-march armclang option
-mcmse armclang option
-S armclang option
--target armclang option
__attribute__((cmse_nonsecure_entry)) function attribute
SG instruction
--cpu armlink option
--import_cmse_lib_in armlink option
--import_cmse_lib_out armlink option
--scatter armlink option
--text fromelf option

13.7 Armv8.1-M PACBTI extension mitigations against ROP and JOP style attacks

The Arm® Compiler for Embedded FuSa support of the Armv8.1-M PACBTI extension mitigates against a number of attacks.

Note

This topic describes a [BETA] feature. See Support level definitions.

The Armv8.1-M PACBTI extension consists of the following control-flow integrity approaches:

- Return address signing and authentication (PAC-RET) mitigates against Return Oriented Programming (ROP) style attacks.
- `BTI` instruction placement (BTI) mitigates against Jump Oriented Programming (JOP) style attacks and restricts the set of targets for an indirect branch.

For more information about ROP and JOP style attacks, see Learn the architecture: Providing protection for complex software.

To use the PACBTI feature effectively:

- Your code must initialize the features and the keys.
- Your code must obtain a sufficiently random initial seed for a random key at program startup. For example, do not use the date and time because an attacker can replicate them.

Startup initialization

If a source of true randomness is available, you must use it to select a random encryption key to initialize PAC. Otherwise, you can use the following sequence for testing only:

The following sequence must only be used for testing.

```
// Set up a PAC signing key.
movw    r2,#0xfb42
movt    r2,#0x11e7
msr     PAC_KEY_P_0,r2
movw    r2,#0xeea2
movt    r2,#0xfc6f
msr     PAC_KEY_P_1,r2
movw    r2,#0xc231
movt    r2,#0x02c7
msr     PAC_KEY_P_2,r2
movw    r2,#0x6582
movt    r2,#0xa269
msr     PAC_KEY_P_3,r2

// CONTROL register: set PAC_EN to enable PAC in privileged mode.
mrs     r2,CONTROL
orr     r2,r2,#0x00000040
msr     CONTROL,r2
```

Enable BTI as follows:

```
// CONTROL register: set BTI_EN, to enable BTI in privileged mode.
mrs     r2,CONTROL
orr     r2,r2,#0x00000010
msr     CONTROL,r2
```

EABI build attributes

The following build attributes have been added to indicate the PACTBI-M features used when compiling code:

Table 13-3: PACRET-M build attributes

Build attribute	<tag>	Value and meaning
Tag_PAC_extension	50	0 - The user did not permit this entity to use PAC/AUT instructions. 1 - The user permitted this entity to use PAC/AUT instructions in the hint space. 2 - The user permitted this entity to use PAC/AUT instructions in the hint and in the non-hint space.
Tag_BTI_extension	52	0 - The user did not permit this entity to use BTI instructions. 1 - The user permitted this entity to use BTI instructions in the hint space. 2 - The user permitted this entity to use BTI instructions in the hint and in the non-hint space.
Tag_BTI_use	74	0 - This code is compiled without branch target enforcement. 1 - This code is compiled with branch target enforcement.
Tag_PACRET_use	76	0 - This code is compiled without return address signing and authentication. 1 - This code is compiled with return address signing and authentication.

When compiling with `-mbranch-protection=pac-ret`, the compiler emits:

```
    .eabi_attribute Tag_PAC_extension, 1
    .eabi_attribute Tag_PACRET_use, 1
```

When compiling with `-mbranch-protection=bti`, the compiler emits:

```
    .eabi_attribute Tag_BTI_extension, 1
    .eabi_attribute Tag_BTI_use, 1
```

The output of PACBTI build attributes depends only on the command-line options given. The build attributes are not affected by function attributes.

These attributes are output only when compiling C or C++ source. They are not output for assembly files. If you are linking with objects that are compiled with a PACBTI feature enabled, we recommend that you add the following code to your assembly language source files:

```
#if !defined(__ARM_64BIT_STATE)
#ifdef __ARM_FEATURE_PAC_DEFAULT
    .eabi_attribute Tag_PAC_extension, 1
    .eabi_attribute Tag_PACRET_use, 1
#endif
#ifdef __ARM_FEATURE_BTI_DEFAULT
    .eabi_attribute Tag_BTI_extension, 1
    .eabi_attribute Tag_BTI_use, 1
#endif
#endif
```

If the assembly source uses non-hint-space PACBTI instructions, you must change the directive for the PAC extension to:

```
    .eabi_attribute Tag_PAC_extension, 2
```

Note

Without these directives, you might report an incompatible build attributes error.

Linker behavior

The following table shows the linker behavior for objects compiled with the Armv8.1-M PACBTI feature and `-mbranch-protection` options:

Note

The same attributes are generated for each `-mbranch-protection` option with or without specifying the `+pacbti` feature.

Note

There is only one library variant for the Armv8.1-M PACBTI extension. This variant provides both pointer authentication and BTI. It is not possible to specify a library variant that supports only one or the other.

Table 13-4: Build attributes and linker behavior

armclang option	Build attribute	Interpretation	Linker behavior
`-mbranch-protection=bti`	`Tag_BTI_use`	Use BTI and link to the Armv8.1-M PACBTI libraries.	The linker issues a warning about mixing BTI with non-BTI objects, for objects that you explicitly specify on the command-line or from user libraries. If the `--require-bti` linker option is specified, an error is issued instead of a warning.
`-mbranch-protection=pac-ret`	`Tag_PACRET_use`	Use PAC-RET and link to the Armv8.1-M PACBTI libraries.	The linker allows mixing PAC-RET with non-PAC-RET objects.
`-mbranch-protection=bti+pac-ret`	`Tag_PACRET_use`, `Tag_BTI_use`	Use BTI and PAC-RET and link to the Armv8.1-M PACBTI libraries.	The linker allows mixing PAC-RET with non-PAC-RET objects.

You can override this behavior by using the linker option `--library_security=<option>`, as shown in the following table:

Table 13-5: --library_security options and linker behavior

armlink option	Linker behavior
`--library_security=none`	Forces the linker to select a non-PACBTI library and suppresses warnings and errors about mixing BTI and non-BTI user objects. For example, where the linker would have selected `c_xua.l`, passing `--library_security=none` would make the linker select either `c_xu.l` or `c_wu.l` depending on final product.
`--library_security=pacbti-m`	Forces the linker to always select an Armv8.1-M PACBTI library and suppress errors about mixing BTI and non-BTI user objects.

You can use the linker option `--info=bti` to output a list of the BTI and non-BTI user objects in the link.

Related information

-march
-mbranch-protection
-mcpu
__attribute__((target("options"))) function attribute
--info=topic[,topic,...] (armlink)
--library-security-protection
--require-bti

13.8 Overview of the Realm Management Extension

The Realm Management Extension (RME) is an extension to the Arm®v9-A application profile architecture. RME provides support for confidential computing environments, known as Realms.


Note

The RME support level is [ALPHA]. See Support level definitions.

RME adds the following features:

- Two additional Security states, Root and Realm.
- Two additional physical address spaces, Root and Realm.
- The ability to dynamically transition memory granules between physical address spaces.
- Granule Protection Check mechanism.


Note

RME does not have an associated `+[no]<feature>` option for the `-march` or `-mcpu` options, because the RME registers are available in the Armv9-A application profile architecture without an additional extension.

For more information, see:

- Introducing Arm Confidential Compute Architecture.
- Arm Confidential Compute Architecture software stack.
- Learn the architecture: Realm Management Extension.
- The Realm Management Extension (RME), for Armv9-A.

13.9 Overview of memory tagging

Memory tagging stack protection (stack memory tagging) and heap memory tagging are available for the AArch64 state for architectures with the Memory Tagging Extension (MTE), `+memtag`. MTE is optional in Arm®v8.5-A and later architectures.

Requirements when using memory tagging

You must be aware of the following requirements when using memory tagging:

- When using the `armclang` option `-fsanitize=memtag-stack` to enable memory tagging on the stack, you must make sure to place the stack in tagged memory.
- When using the `armclang` option `-fsanitize=memtag-heap` to enable memory tagging on the heap, you must make sure to place the heap in tagged memory.

- When defining the symbol `__use_memtag_heap` to enable the heap implementation that uses memory tagging, you must make sure to place the heap in tagged memory.
- You must ensure that the tagged memory used for the stack and heap has an initial tag value of zero.

Stack memory tagging

Use `-fsanitize=memtag-stack` to enable the generation of memory tagging code for protecting the memory allocations on the stack. The resulting code cannot execute on architectures without the MTE. For more information, see the `+memtag` feature in -mcpu.

When you enable memory tagging, the compiler checks that expressions that evaluate to addresses of objects on the stack are within the bounds of the object. If this cannot be guaranteed, the compiler generates code to ensure that the pointer and the object are tagged. When tagged pointers are dereferenced, the processor checks the tag on the pointer with the tag on the memory location being accessed. If the tags do not match, the processor generates an exception and therefore tries to prevent the pointer from accessing any object that is different from the object whose address was taken.

For example, if a pointer to a variable on the stack is passed to another function, then the compiler might be unable to guarantee that this pointer is only used to access the same variable. In this situation, the compiler generates memory tagging code. The memory tagging instructions apply a unique tag to the pointer and to its corresponding allocation on the stack.


Note

- The ability of the compiler to determine whether a pointer access is bounded might be affected by optimizations. For example, if an optimization inlines a function, and as a result, if the compiler can guarantee that the pointer access is always safe, then the compiler might not generate memory tagging stack protection code. Therefore, the conditions for generating memory tagging stack protection code might not have a direct relationship to the source code.
- When using `-fsanitize=memtag-stack`, there is a high probability that an unbounded pointer access to the stack causes a processor exception. This option does not guarantee that all unbounded pointer accesses to the stack cause a processor exception.
- The implementation of stack tagging does not protect variable-length allocations on the stack.
- Use of `-fsanitize=memtag-stack` to protect the stack increases the amount of memory that is allocated on the stack. This memory increase is because the compiler has to allocate a separate 16-byte aligned block of memory on the stack for each variable whose stack allocation is protected by memory tagging.
- Code that is compiled with stack tagging can be safely linked together with code that is compiled without stack tagging. However, if any object file is compiled with `-fsanitize=memtag-stack`, and if `setjmp`, `longjmp`, or C++ exceptions are present anywhere in the image, then you must use the `v8.5a` library to avoid stack tagging related memory fault at runtime.

- The `-fsanitize=memtag-stack` option and the `-fstack-protector` options are independent and provide complementary stack protection. These options can be used together or in isolation.

Heap memory tagging

Heap memory tagging protects against heap overflow attacks. To access this protection mechanism, use the `armclang` option `-fsanitize=memtag-heap` and define the `armclang` symbol `__use_memtag_heap`. `-fsanitize=memtag-heap` makes code generation changes for Armv8.5-A and later targets that support the Memory Tagging Extension (MTE) extension to protect against heap overflow attacks. `__use_memtag_heap` makes the linker select heap functions in the library that have memory tagging enabled. For more information, see Choosing a heap implementation for memory allocation functions.

Library support

To ensure full memory tagging protection, you must also link your code with the library that provides memory tagging protection. For more information, see armlink --library_security=protection.

`armlink` automatically selects the library with memory tagging protection if at least one object file is compiled with pointer authentication using `-mbranch-protection`, and one of the following is true:

- At least one object file is compiled with `-fsanitize=memtag-stack`.
- At least one object file includes the symbol `__use_memtag_heap` and is compiled with `-fsanitize=memtag-heap`.

You can override the selected library by using the `armlink` option `--library_security` to specify the library that you want to use.

Related information

armclang -fsanitize, -fno-sanitize
armclang -fstack-protector, -fstack-protector-all, -fstack-protector-strong, -fno-stack-protector
armclang -mbranch-protection
armclang -mcpu
armlink --library_security=protection
Choosing a heap implementation for memory allocation functions
Arm C Language Extensions

13.10 Overview of Control Flow Integrity

Control Flow Integrity (CFI) sanitizer implements a number of CFI schemes. These schemes are designed to abort the program on detection of certain forms of undefined behavior that can potentially allow attackers to subvert the control flow of the program.

The CFI schemes are:

Table 13-6: Control Flow Integrity schemes supported

Scheme	Description
`cfi-cast-strict`	Enables strict cast checks.
`cfi-derived-cast`	Base-to-derived cast to the wrong dynamic type.
`cfi-unrelated-cast`	Cast from `void*` or another unrelated type to the wrong dynamic type.
`cfi-nvcall`	Non-virtual call through an object that has a `vptr` of the wrong dynamic type.
`cfi-vcall`	Virtual call through an object that has a `vptr` of the wrong dynamic type.
`cfi-icall`	Indirect call of a function with wrong dynamic type.
`cfi-mfcall`	Indirect call through a member function pointer with wrong dynamic type.

You can enable any of the CFI schemes individually, or enable all schemes with `-fsanitize=cfi` then disable some of them with the `-fno-sanitize` option. For example, to disable the `cfi-nvcall` and `cfi-icall` schemes, specify:

```
-fsanitize=cfi -fno-sanitize=cfi-nvcall,cfi-icall -fvisibility=hidden
```

If you enable at least one CFI scheme with `-fsanitize`, then you must also enable Link-Time Optimization (LTO) with the `armclang` option `-flto` and the `armlink` option `--lto`.

CFI also uses an ignore list that is a list of entities for which the CFI checks are to be relaxed. This list is populated from a text file `cfi_ignorelist.txt`. Arm® Compiler for Embedded FuSa provides an empty `cfi_ignorelist.txt` file. By default, `armclang` searches for this file in `<install_path>/lib/clang/<version>/share`:

- You can change the default location that `armclang` searches for the `cfi_ignorelist.txt` file with the `-resource-dir=<path_to_resource_folder>` option.
- If you want to clear the ignore list, then specify the `armclang` option `-fno-sanitize-ignorelist`.
- If you want to extend the ignore list using your own ignore list files, then specify each file with `-fsanitize-ignorelist=<ignorelistfile>`.

The member function pointer call checking scheme, `cfi-mfcall`, checks to make sure that the base type of the member function pointer is complete. `armclang` only emits a full CFI check if this base type is complete. To ensure `armclang` always emits a full CFI check, you must specify `-fcomplete-member-pointers`.

For more information about the CFI checks, see Control Flow Integrity.

Note

Arm Compiler for Embedded FuSa does not support the `-flto=thin` and `-fno-sanitize-trap` options.

See also *List of known unsupported features* in Support level definitions.

Related information

Support level definitions on page 414
armclang -fcomplete-member-pointers
armclang -fsanitize, -fno-sanitize
armclang -fsanitize-ignorelist, -fno-sanitize-ignorelist
armclang -resource-dir
armclang -flto, -fno-lto
armlink -lto, -no_lto

13.11 Overview of Undefined Behavior Sanitizer

The Undefined Behavior Sanitizer (UBSan) is a code instrumentation inserted by the compiler to catch undefined behaviors during runtime.

UBSan has the following modes:

Traps mode
Execute trap instructions on undefined behavior detection.

Minimal handlers mode
Call minimal handlers on undefined behavior detection.

Non-minimal handlers mode
Call regular handlers on undefined behavior detection. Arm® Compiler for Embedded FuSa does not support this mode.

To catch a particular kind of Undefined Behavior, specify the required check with the `armclang` option `-fsanitize=<ubsan_check>`. For a complete list of checks, see *Available checks* at Undefined Behavior Sanitizer.

However, the option `-fsanitize=undefined` enables all the UBSan checks, except for `float-divide-by-zero`, `unsigned-integer-overflow`, `implicit-conversion`, `local-bounds`, and the `nullability-*` group of checks. To prevent the non-minimal handlers mode from being enabled, you must include checks that relate to the traps mode and the minimal handlers mode:

- To enable the traps mode for a particular check, specify the required check with the `armclang` option `-fsanitize-trap=<ubsan_check>`. Alternatively, you can specify `-fsanitize-trap=all` to use traps mode for all checks requested.
- To enable the minimal handlers mode, specify the `armclang` option `-fsanitize-minimal-runtime`.

Related information

armclang -fsanitize, -fno-sanitize
armclang -fsanitize-minimal-runtime
armclang -fsanitize-trap, -fno-sanitize-trap
armclang -fsanitize-recover, -fno-sanitize-recover
armclang -fstrict-flex-arrays

13.12 Overview of Straight-Line Speculation hardening

Some processors might speculatively execute the instructions immediately following changes in control flow, including RET (returns), BR (indirect jumps), and BLR (indirect function calls). If the speculative execution path contains a suitable code sequence, such Straight-Line Speculation (SLS) could lead to changes in the caches and similar structures that contain secrets, making those secrets vulnerable to revelation through timing analysis.

The `armclang` option `-mharden-sls=<option>` allows you to mitigate against this vulnerability.

For RET and BR instructions, the mitigation places a speculation barrier after the instructions that prohibits incorrect speculation. `armclang` uses the SB speculation barrier instruction after RET and BR instructions if that instruction is supported by the target. Otherwise, it uses the DSB and ISB instructions.

For the BLR instruction, the mitigation replaces all instances of BLR with a BL and BR sequence, for example:

```
    blr x<N>
```

This instruction gets transformed to:

```
    bl __llvm_slsblr_thunk_x<N>
```

`armclang` creates a thunk `__llvm_slsblr_thunk_x<N>` for every `x<N>` register. Each thunk is placed in a separate section named `.text.__llvm_slsblr_thunk_x<N>` that contains:

```
    .section
  .text.__llvm_slsblr_thunk_x<N>,"axG",@progbits,__llvm_slsblr_thunk_x<N>,comdat
    .hidden __llvm_slsblr_thunk_x<N>          // -- Begin function
 __llvm_slsblr_thunk_x<N>
    .weak    __llvm_slsblr_thunk_x<N>
    .p2align    4
    .type    __llvm_slsblr_thunk_x<N>,@function
__llvm_slsblr_thunk_x<N>:
    br x<N>
    dsb sy
    isb
```


Note

The register number in the thunk might be different from the register in the original BLR instruction.

The BLR instruction gets split into separate BL and BR instructions. This transformation results in not inserting a speculation barrier on the architectural execution path.

In Arm® Compiler for Embedded FuSa 6, the separate thunk code is globally visible and might be called from a location where the `SB` instruction is locally disabled. Therefore, `armclang` always uses the `DSB` and `ISB` speculation barrier instructions.

The linker unused section elimination feature removes all unused thunk sections. Also, these sections are generated in every object file included in the compile. Because the sections are defined in comdat groups, the linker includes only one instance in the output.

Placement of .text.__llvm_slsblr_thunk_x<N> sections

You can place the `.text.__llvm_slsblr_thunk_x<N>` sections with a scatter file as follows:

```
    *(.text.__llvm_slsblr_thunk_x*)
```

If you place the sections far away from the references, the linker adds a veneer to locate them.

Restrictions of SLS hardening

SLS hardening is supported on AArch64 targets, but is not available in the following situations:

- Use of `BR`, `RET`, and `BLR` instructions in assembly code.
- Use of `BR`, `RET`, and `BLR` instructions in libraries and run-time library routines that are not recompiled with this toolchain mitigation.

We do not provide compiler-generated mitigations for all the other instructions mentioned in the Straight-line speculation whitepaper.

Related information

-mharden-sls
BLR instruction
BR instruction
DSB instruction
ISB instruction
RET instruction
SB instruction

13.13 Memory-safety best practices

Memory-safety is an important aspect of security hardening that addresses issues such as using uninitialized memory, buffer overflow, and use after free.

The following techniques are recommended to improve memory-safety of C and C++ code:

Develop code following coding guidelines

There are industry accepted guidelines such as MISRA, AUTOSAR, CERT, and C++ Core Guidelines. Particularly, C++ guidelines focus on avoiding or encapsulating the use of raw pointers and arrays by replacing them with smart pointers and standard C++ library containers.

C++ provides more safety features than C. Therefore C++ might be a better choice for projects where safety is important and technical constraints allow use of C++.

Visit Carnegie Mellon University and search for the following titles:

- *SEI CERT C++ Coding Standard*.
- *SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems (2016 Edition)*.

Perform static analysis

Commercial and open source third-party tools, such as the LLVM Project `clang-tidy`, are available. Such tools allow you to get the most thorough analysis of the code. This analysis includes checking for the compliance with coding guidelines. Arm® Compiler for Embedded FuSa provides a set of analysis and associated warnings such as:

- `-Wall` and `-Wextra`.
- `-Wformat=2`.

Use memory-safety hardening build options

Arm Compiler for Embedded FuSa provides the following features to address various memory-safety issues:

- -mbranch-protection=standard. This option requires the Pointer Authentication architecture extension. See Armv8.1-M PACBTI extension for more information.
- -fstack-protector-strong.
- -fsanitize=shadow-call-stack.
- -fsanitize=memtag-stack and -fsanitize=memtag-heap. These options require the memory tagging architecture extension. See Overview of memory tagging for more information.
- -ftrivial-auto-var-init.

Perform dynamic analysis

Arm Compiler for Embedded FuSa provides the Undefined Behavior Sanitizer (UBSan) feature to sanitize code and catch memory-safety defects at run-time during testing.

Third-party fuzz testing tools are available to improve code coverage during testing. These tools help you to find more memory-safety issues. Third-party bounded model checking tools can verify memory-safety properties, among other properties, by using formal proof methods.

Note

There is a wider choice of tools for dynamic analysis on host operating systems than on embedded systems. Therefore, structuring code in a way that allows the running of application logic tests on a host operating system can provide more opportunities to use dynamic analysis tools.

Related information

Security features supported in Arm Compiler for Embedded FuSa on page 285
-W (armclang)

14. Thread-Local Storage

Thread-Local Storage (TLS) is a method of managing memory in systems that have separate threads. TLS uses static or global memory that is local and unique to a thread. A single instance of a static or global variable is allocated for each thread that exists.


Note

This topic includes descriptions of [ALPHA] and [COMMUNITY] features. See Support level definitions.

When using multiple threads, each thread you create must have an instance of the TLS data area. On switching context, you must arrange for the thread pointer to point to the TLS data area for the thread.

Many functions in the standard C library use a persistent state in the library. For example, the global variable `errno` holds the error status from the library, so it must not be overwritten by other threads.

The standard C library startup code and linker scatter file ensures that the TLS is instantiated once for the main thread, for use in single threaded systems.


Note

TLS is a replacement for the __user_libspace static data area.

TLS support in C and C++

Arm® Compiler for Embedded FuSa provides TLS support in C and C++ as follows:

TLS in C

Arm Compiler for Embedded FuSa supports the `__thread` storage class keyword in C.

For C multithreaded support in Arm Compiler for Embedded FuSa, see Multithreaded support in Arm C libraries.

TLS in C++

Arm Compiler for Embedded FuSa supports the `__thread` storage class keyword in C++.

Arm Compiler for Embedded FuSa supports the thread storage duration specifier of C++, `thread_local`, for `-std=c++11` or later. This keyword is only supported for C in Arm Compiler for Embedded FuSa version 6.19 and later when used with the `-std=c2x` [COMMUNITY] feature for C23 support.

For C++ multithreaded support in Arm Compiler for Embedded FuSa, see Multithreaded support in Arm C++ libraries. The Arm C++ libraries support level for multithreaded applications is [ALPHA].

TLS models supported in Arm Compiler for Embedded FuSa

Various TLS models are supported that specify how access to variables is handled. You can specify the model as follows:

- For a complete compilation, specify the `-ftls-model` compiler option. For more information, see -ftls-model.
- For a specific variable, use the `__attribute__((tls_model("model")))` variable attribute. For more information, see __attribute__((tls_model("model"))) variable attribute.

If you specify the `-ftls-model=<model>` command-line option and your code includes the `__attribute__((tls_model("<model>")))` variable attribute, then the attribute overrides the command-line option.

For executables that are statically linked, you need only use the `local-exec` model. `local-exec` is the least general and most efficient model.

For dynamic linking, you might consider using the `initial-exec` and `global-dynamic` models. However, the compiler always selects a compatible model:

- `global-dynamic` is the most general but least efficient model, and you can use it anywhere.
- You can use the `initial-exec` model in shared-libraries provided that they are not loaded at runtime with `dlopen()`.

Arm Compiler for Embedded FuSa supports TLS in the following linking models:

- Thread-Local Storage in the bare metal and shared library linking models.
- Thread-Local Storage in the SysV linking model.

Example: TLS example for AArch64

AArch64 TLS local-exec static linking example describes an example with source code that you can build and run.

14.1 AArch64 TLS local-exec static linking example

This example is based on the `startup_Armv8-Ax1_AC6` example provided with Arm® Development Studio. It can run on a single-core. We have tested it with the `FVP_Base_Cortex-A53x1` model that is shipped with Arm Development Studio.

Note

This example is not a complete solution and is provided only to show the Thread-Local Storage (TLS) features available in Arm Compiler for Embedded FuSa.

What the example does

This example does the following:

- Places TLS RW and ZI data in a specific location in memory using a scatter file.

- At the start of the `main()` function, initializes TLS data by using linker-defined symbols to find the data in memory. You must use the equivalent symbols for your own implementation.
- Accesses the initialized TLS data and prints it to the terminal.

Requirements for using AArch64 TLS local-exec with static linking

The following are the requirements for using AArch64 TLS `local-exec` with static linking:

- Annotate the TLS RW and ZI variables with the `__thread` or `_thread_local` keyword. This example uses `__thread`.
- Either compile with the `-ftls-model=local-exec` option or annotate TLS RW and ZI variables with the `__attribute__((tls_model("local-exec")))` variable attribute.
- Provide an implementation of the `void write_tp(void* tls_data)` function, This function writes the pointer `tls_data` to the `TPIDR_ELn` register that you want to use. This example provides the following definition in `main.c`:

```
__attribute__((always_inline)) static void write_tp(void* tls_data)
{
    __asm volatile("msr TPIDR_EL0, %0" : : "r"(tls_data) : "cc");
}
```

- Compile with `-mtp=<el>` to specify the `TPIDR_ELn` register to use. For example, to use `TPIDR_EL0`, compile with `-mtp=el0`.
- Place the TLS RW and ZI data using a scatter file in the following order of increasing addresses:
  - If the TLS RW and ZI data is part of an existing load region:
    1. Any RO code and data as needed.
    2. TLS RW data.
    3. No gaps, other than alignment padding.
    4. TLS ZI data.
    5. Any non-TLS RW and ZI data as needed.
  - If the TLS RW and ZI data is in its own dedicated load region:
    1. TLS RW data.
    2. Make sure there are no gaps other than alignment padding.
    3. TLS ZI data.

  You can use the `+tls-rw` selector to select the TLS RW data. You can use the `+tls-zi` selector to select the TLS ZI data. You must keep all the TLS data for the entire application in one execution region.

  This example places the TLS RW and ZI data in an existing load region called `LOAD`:

```
LOAD 0x80000000
{
    STARTUP +0
    {
        startup.o (StartUp, +FIRST)
    }
```

```
    EXEC +0 {
        *(+RO, +RW, +ZI)
    }

    ;
    ; TLS RW region
    ; If the load region contains more execution regions
    ; than just TLS execution regions, then do not place
    ; any non-TLS RW or ZI data before TLS RW or ZI data
    ;
    ER_TLS_RW +0 {
        *(+tls-rw)
    }

    ;
    ; TLS ZI region
    ; This must be immediately after the TLS RW region
    ;
    ER_TLS_ZI +0 {
    *(+tls-zi)
    }
...
}
```

- Provide your own implementation of a function that initializes the TLS RW and ZI data for each thread from its initial location in memory.

  If your TLS RW data is in an execution region `ER_TLS_RW` and your TLS ZI data is in an execution region `ER_TLS_ZI`, then you can use the following linker-defined symbols to determine the TLS data attributes:

```
// Start address of TLS RW data
(unsigned int*)&Image$$ER_TLS_RW$$Base

// Number of bytes of TLS RW data
(size_t)&Image$$ER_TLS_RW$$Limit - (size_t)&Image$$ER_TLS_RW$$Base

// Number of bytes of TLS ZI data
// Using this calculation takes into account
// any alignment padding between the
// ER_TLS_RW and ER_TLS_ZI execution regions
(size_t)&Image$$ER_TLS_ZI$$ZI$$Limit - (size_t)&Image$$ER_TLS_RW$$Limit
```

  Note

  It is important that you use `$$ZI$$` when referring to the TLS ZI data. Without it, the linker does not include the ZI data when calculating the value of the linker-defined symbol.

  For more information, see linker-defined symbols.

- Link with the `--bare_metal_sysv` and `--sysv` options.

Requirements for building and running the example

To build the example, do the following:

1. Create a project folder to contain the build and source files, for example `tls-aarch64-scatter-loading-example`.

2. Create the build, clean, and run scripts for your environment and place them in the project folder. See the following for the contents of the scripts:
   - Build and clean scripts for the AArch64 TLS local-exec static linking example.
   - Run scripts for the AArch64 TLS local-exec static linking example.
3. Create the scatter file shown in Scatter file for the AArch64 TLS local-exec static linking example, and place it in the project folder.
4. Create the `asm`, `src`, and `obj` folders in the project folder.
5. Create the assembly source files shown in Assembly source files for the AArch64 TLS local-exec static linking example, and place them in the `asm` folder.
6. Create the C source files shown in C source files for the AArch64 TLS local-exec static linking example, and place them in the `src` folder.

Building and running the example

You can use the build scripts `build.sh` and `build.bat` to build the example on a Linux or Windows environment respectively. The scripts generate the image file `tls_aarch64.axf`.

You can use the `run.sh` and `run.bat` scripts to run the example on the `FVP_Base_Cortex-A53x1` FVP that is shipped with Arm Development Studio. You must provide the path to the directory containing the FVP executable when running these scripts. For example, with Arm Development Studio 2021.0 installed to the default installation directory:

- On Linux:

```
./run.sh /opt/arm/developmentstudio-2021.0/sw/models/bin
```

- On Windows:

```
run.bat "\Program Files\Arm\Development Studio 2021.0\sw\models\bin"
```

When you run the example, it prints messages similar to the following:

```
TLS RW foo @ 0x0000000080051480 = 0xdeadbeef
TLS ZI bar @ 0x0000000080051484 = 0x0
```

The addresses are from after TLS data initialization at run-time. You can verify that the RW address is not the link-time address of the TLS RW data by examining the memory map in the `tls_aarch64.lst` file. For example:

```
    Execution Region ER_TLS_RW (Exec base: 0x80004250, Load base: 0x80004250, Size:
0x00000004, Max: 0xffffffffffffffff, ABSOLUTE)

    Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name
     Object

    0x80004250   0x80004250   0x00000004   Data   RW           45    .tdata.foo
     main.o


    Execution Region ER_TLS_ZI (Exec base: 0x80004254, Load base: 0x80004254, Size:
0x00000000, Max: 0xffffffffffffffff, ABSOLUTE)
```

```
    Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name
     Object

    0x80004254        -       0x00000004   Zero   RW          46    .tbss.bar
     main.o
```

Related information

-ftls-model
-mtp
__attribute__((tls_model("model"))) variable attribute
--sysv
--bare_metal_sysv
Requirements and restrictions for using scatter files with SysV linking model

14.2 Build and clean scripts for the AArch64 TLS local-exec static linking example

The build script provides the `armclang` and `armlink` commands to build the Thread-Local Storage (TLS) example. Use the clean script to remove the files generated by these commands. There is a build and clean script for both Windows and Linux environments.

build.sh and clean.sh scripts for Linux environments

Create the `build.sh` script containing the following commands:

```
# Compile files from Arm DS example
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src/
retarget.c -o obj/retarget.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src/
uart.c -o obj/uart.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src/
GICv3_gicd.c -o obj/GICv3_gicd.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src/
GICv3_gicr.c -o obj/GICv3_gicr.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src/
sp804_timer.c -o obj/sp804_timer.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src/
timer_interrupts.c -o obj/timer_interrupts.o

# Assemble startup files from Arm DS example
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -c asm/
startup.S -o obj/startup.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -c asm/
vectors.S -o obj/vectors.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -c asm/
v8_utils.S -o obj/v8_utils.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -c asm/
v8_aarch64.S -o obj/v8_aarch64.o

# Compile example single-threaded TLS code
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src/
main.c -o obj/main.o -mtp=el0

# Link everything
armlink --cpu=8-A.64 --sysv --bare_metal_sysv --scatter=scatter.scat --
diag_suppress=6329 --entry start64 --map --load_addr_map_info --list tls_aarch64.lst
```

```
 obj/retarget.o obj/uart.o obj/GICv3_gicr.o obj/main.o obj/sp804_timer.o obj/
timer_interrupts.o obj/GICv3_gicd.o obj/startup.o obj/v8_utils.o obj/v8_aarch64.o
 obj/vectors.o -o tls_aarch64.axf
```

Create the `clean.sh` script containing the following commands:

```
rm obj/*
rm tls_aarch64.lst
rm tls_aarch64.axf
```

build.bat and clean.bat scripts for Windows environments

Create the `build.bat` script containing the following commands:

```
REM Compile files from Arm DS example
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src
\retarget.c -o obj\retarget.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src
\uart.c -o obj\uart.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src
\GICv3_gicd.c -o obj\GICv3_gicd.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src
\GICv3_gicr.c -o obj\GICv3_gicr.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src
\sp804_timer.c -o obj\sp804_timer.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src
\timer_interrupts.c -o obj\timer_interrupts.o

REM Assemble startup files from Arm DS example
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -c asm
\startup.S -o obj\startup.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -c asm
\vectors.S -o obj\vectors.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -c asm
\v8_utils.S -o obj\v8_utils.o
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -c asm
\v8_aarch64.S -o obj\v8_aarch64.o

REM Compile example single-threaded TLS code
armclang --target=aarch64-arm-none-eabi -march=armv8-a -Isrc -DCORTEXA -O1 -c src
\main.c -o obj\main.o -mtp=el0

REM Link everything
armlink --cpu=8-A.64 --sysv --bare_metal_sysv --scatter=scatter.scat --
diag_suppress=6329 --entry start64 --map --load_addr_map_info --list tls_aarch64.lst
 obj\retarget.o obj\uart.o obj\GICv3_gicr.o obj\main.o obj\sp804_timer.o obj
\timer_interrupts.o obj\GICv3_gicd.o obj\startup.o obj\v8_utils.o obj\v8_aarch64.o
 obj\vectors.o -o tls_aarch64.axf
```

Create the `clean.bat` script containing the following commands:

```
del obj\*
del tls_aarch64.lst
del tls_aarch64.axf
```

14.3 Run scripts for the AArch64 TLS local-exec static linking example

Create the run scripts for your environment.

run.sh script for Linux environments

Create the `run.sh` script containing the following command:

```
$1/FVP_Base_Cortex-A53x1 -C bp.secure_memory=false -C bp.vis.disable_visualisation=1
 -a tls_aarch64.axf
```

run.bat script for Windows environments

Create the `run.bat` script containing the following command:

```
%1\FVP_Base_Cortex-A53x1 -C bp.secure_memory=false -C bp.vis.disable_visualisation=1
 -a tls_aarch64.axf
```

14.4 Scatter file for the AArch64 TLS local-exec static linking example

Create the scatter file `scatter.scat` for the AArch64 TLS local-exec static linking example.

```
;*************************************************************
; Scatter file for Armv8-A Startup code on FVP Base model
; Copyright (c) 2014-2016 Arm Limited (or its affiliates). All rights reserved.
; Use, modification and redistribution of this file is subject to your possession
; of a valid End User License Agreement for the Arm Product of which these
; examples are part of and your compliance with all applicable terms and
; conditions of such licence agreement.
;*************************************************************

LOAD 0x80000000
{
    STARTUP +0
    {
        startup.o (StartUp, +FIRST)
    }

    EXEC +0 {
        *(+RO, +RW)
    }

    ;
    ; TLS RW region
    ; If the load region contains more execution regions
    ; than just TLS execution regions, then do not place
    ; any non-TLS RW or ZI data before TLS RW or ZI data
    ;
    ER_TLS_RW +0 {
        *(+tls-rw)
    }

    ;
    ; TLS ZI region
```

```
    ; This must be immediately after the TLS RW region
    ;
    ER_TLS_ZI +0 {
      *(+tls-zi)
    }

    ER_ZI +0 {
        *(+ZI)
    }

    ;
    ; GICv3 distributor
    ;
    GICD +0 UNINIT 0x8000
    {
        GICv3_gicd.o (.bss.distributor)
    }

    ;
    ; GICv3 redistributors
    ; 128KB for each redistributor in the system
    ;
    GICR +0 UNINIT 0x80000
    {
        GICv3_gicr.o (.bss.redistributor)
    }

    ;
    ; App stack
    ; All stacks and heap are aligned to a cache-line boundary
    ;
    ARM_LIB_STACK    +0 ALIGN 64 EMPTY 0x4000 {}

    ;
    ; Stack for EL3
    ;
    EL3_STACKS       +0 ALIGN 64 EMPTY 0x1000 {}

    ;
    ; Separate heap - import symbol __use_two_region_memory
    ; in source code for this to work correctly
    ;
    ARM_LIB_HEAP     +0 ALIGN 64 EMPTY 0xA0000 {}

    ;
    ; Strictly speaking, the L1 tables do not need to
    ; be so strongly aligned, but no matter
    ;
    TTB0_L1          +0 ALIGN 4096 EMPTY 0x1000 {}

    ;
    ; Various sets of L2 tables
    ;
    ; Alignment is 4KB, since the code uses a 4K page
    ; granularity - larger granularities would require
    ; correspondingly stricter alignment
    ;
    TTB0_L2_RAM      +0 ALIGN 4096 EMPTY 0x1000 {}

    TTB0_L2_PRIVATE  +0 ALIGN 4096 EMPTY 0x1000 {}

    TTB0_L2_PERIPH   +0 ALIGN 4096 EMPTY 0x1000 {}

    ;
    ; The startup code uses the end of this region to calculate
    ; the top of memory - do not place any RAM regions after it
    ;
    TOP_OF_RAM  +0 EMPTY 4 {}

    ;
    ; CS3 Peripherals is a 64MB region from 0x1c000000
```

```
    ; that includes the following:
    ; System Registers             at 0x1C010000
    ; UART0 (PL011)                at 0x1C090000
    ; Color LCD Controller (PL111) at 0x1C1F0000
    ; plus a number of others.
    ; CS3_PERIPHERALS is used by the startup code for page-table generation
    ; This region is not truly empty, but we have no
    ; predefined objects that live within it
    ;
    CS3_PERIPHERALS  0x1c000000 EMPTY 0x90000 {}

    ;
    ; Place the UART peripheral registers data structure
    ; This is only really needed if USE_SERIAL_PORT is defined, but
    ; the linker will remove unused sections if not needed
    PL011  0x1c090000 UNINIT 0x1000
    {
        uart.o (+ZI)
    }
}
```

14.5 Assembly source files for the AArch64 TLS local-exec static linking example

Create the assembly source files for the AArch64 TLS local-exec static linking example.

List of assembly source files for the example

- `PPM_AEM.h`
- `startup.S`
- `v8_aarch64.S`
- `v8_mmu.h`
- `v8_system.h`
- `v8_utils.S`
- `vectors.S`

Contents of the assembly source files for the example

Create the file `PPM_AEM.h` containing the following code:

```
//
// Private Peripheral Map for the v8 Architecture Envelope Model
//
// Copyright (c) 2012-2017 Arm Limited (or its affiliates). All rights reserved.
// Use, modification and redistribution of this file is subject to your possession
 of a
// valid End User License Agreement for the Arm Product of which these examples are
 part of
// and your compliance with all applicable terms and conditions of such licence
 agreement.
//

#ifndef PPM_AEM_H
#define PPM_AEM_H
```

```
//
// Distributor layout
//
#define GICD_CTLR      0x0000
#define GICD_TYPER     0x0004
#define GICD_IIDR      0x0008
#define GICD_IGROUP    0x0080
#define GICD_ISENABLE  0x0100
#define GICD_ICENABLE  0x0180
#define GICD_ISPEND    0x0200
#define GICD_ICPEND    0x0280
#define GICD_ISACTIVE  0x0300
#define GICD_ICACTIVE  0x0380
#define GICD_IPRIORITY 0x0400
#define GICD_ITARGETS  0x0800
#define GICD_ICFG      0x0c00
#define GICD_PPISR     0x0d00
#define GICD_SPISR     0x0d04
#define GICD_SGIR      0x0f00
#define GICD_CPENDSGI  0x0f10
#define GICD_SPENDSGI  0x0f20
#define GICD_PIDR4     0x0fd0
#define GICD_PIDR5     0x0fd4
#define GICD_PIDR6     0x0fd8
#define GICD_PIDR7     0x0fdc
#define GICD_PIDR0     0x0fe0
#define GICD_PIDR1     0x0fe4
#define GICD_PIDR2     0x0fe8
#define GICD_PIDR3     0x0fec
#define GICD_CIDR0     0x0ff0
#define GICD_CIDR1     0x0ff4
#define GICD_CIDR2     0x0ff8
#define GICD_CIDR3     0x0ffc

//
// CPU Interface layout
//
#define GICC_CTLR      0x0000
#define GICC_PMR       0x0004
#define GICC_BPR       0x0008
#define GICC_IAR       0x000c
#define GICC_EOIR      0x0010
#define GICC_RPR       0x0014
#define GICC_HPPIR     0x0018
#define GICC_ABPR      0x001c
#define GICC_AIAR      0x0020
#define GICC_AEOIR     0x0024
#define GICC_AHPPIR    0x0028
#define GICC_APR0      0x00d0
#define GICC_NSAPR0    0x00e0
#define GICC_IIDR      0x00fc
#define GICC_DIR       0x1000

#endif // PPM_AEM_H
```

Create the file `startup.s` containing the following code:

```
// -----------------------------------------------------------------
// Armv8-A Single-core EL3 AArch64 Startup Code
//
// Basic Vectors, MMU, caches and GICv3 initialization
//
// Exits in EL1 AArch64
//
// Copyright (c) 2014-2020 Arm Limited (or its affiliates). All rights reserved.
// Use, modification and redistribution of this file is subject to your possession
 of a
```

```
// valid End User License Agreement for the Arm Product of which these examples are
 part of
// and your compliance with all applicable terms and conditions of such licence
 agreement.
// -----------------------------------------------------------

#include "v8_mmu.h"
#include "v8_system.h"


    .section StartUp, "ax"
    .balign 4
    .cfi_sections .debug_frame  // put stack frame info into .debug_frame instead
 of .eh_frame

    .global el1_vectors
    .global el2_vectors
    .global el3_vectors

    .global InvalidateUDCaches
    .global ZeroBlock

    .global SetPrivateIntSecurityBlock
    .global SetSPISecurityAll

    .global WakeupGICR
    .global SyncAREinGICD
    .global EnableGICD

    .global __main

    .global Image$$EXEC$$RO$$Base
    .global Image$$TTB0_L1$$ZI$$Base
    .global Image$$TTB0_L2_RAM$$ZI$$Base
    .global Image$$TTB0_L2_PERIPH$$ZI$$Base
    .global Image$$TOP_OF_RAM$$ZI$$Base
    .global Image$$GICD$$ZI$$Base
    .global Image$$ARM_LIB_STACK$$ZI$$Limit
    .global Image$$EL3_STACKS$$ZI$$Limit
    .global Image$$CS3_PERIPHERALS$$ZI$$Base
    // use separate stack and heap, as anticipated by scatter.scat
    .global __use_two_region_memory


// -----------------------------------------------------------

    .global start64
    .type start64, "function"
start64:
    // Extract the core number from MPIDR_EL1 and store it in x19
    // (defined by the AAPCS as callee-saved), so we can re-use it later
    //
    bl GetCPUID
    mov x19, x0

    // If run on a multi-core system, put any secondary cores to sleep
    cbz x19, core0_only
loop_wfi:
    dsb SY      // Clear all pending data accesses
    wfi         // Go to sleep
    b loop_wfi

core0_only:

    //
    // program the VBARs
    //
    ldr x1, =el1_vectors
    msr VBAR_EL1, x1

    ldr x1, =el2_vectors
```

```
    msr VBAR_EL2, x1

    ldr x1, =el3_vectors
    msr VBAR_EL3, x1


    // GIC-500 comes out of reset in GICv2 compatibility mode - first set
    // system register enables for all relevant exception levels, and
    // select GICv3 operating mode
    //
    msr SCR_EL3, xzr  // Ensure NS bit is initially clear, so secure copy of
 ICC_SRE_EL1 can be configured
    isb

    mov x0, #15
    msr ICC_SRE_EL3, x0
    isb
    msr ICC_SRE_EL1, x0 // Secure copy of ICC_SRE_EL1

    //
    // set lower exception levels as non-secure, with no access
    // back to EL2 or EL3, and are AArch64 capable
    //
    mov x3, #(SCR_EL3_RW  | \
              SCR_EL3_SMD | \
              SCR_EL3_NS)      // Set NS bit, to access Non-secure registers
    msr SCR_EL3, x3
    isb

    mov x0, #15
    msr ICC_SRE_EL2, x0
    isb
    msr ICC_SRE_EL1, x0 // Non-secure copy of ICC_SRE_EL1


    //
    // no traps or VM modifications from the Hypervisor, EL1 is AArch64
    //
    mov x2, #HCR_EL2_RW
    msr HCR_EL2, x2

    //
    // VMID is still significant, even when virtualization is not
    // being used, so ensure VTTBR_EL2 is properly initialized
    //
    msr VTTBR_EL2, xzr

    //
    // VMPIDR_EL2 holds the value of the Virtualization Multiprocessor ID. This is
 the value returned by Non-secure EL1 reads of MPIDR_EL1.
    //  VPIDR_EL2 holds the value of the Virtualization Processor ID. This is the
 value returned by Non-secure EL1 reads of MIDR_EL1.
    // Both of these registers are architecturally UNKNOWN at reset, and so they
 must be set to the correct value
    // (even if EL2/virtualization is not being used), otherwise non-secure EL1
 reads of MPIDR_EL1/MIDR_EL1 will return garbage values.
    // This guarantees that any future reads of MPIDR_EL1 and MIDR_EL1 from Non-
secure EL1 will return the correct value.
    //
    mrs x0, MPIDR_EL1
    msr VMPIDR_EL2, x0
    mrs x0, MIDR_EL1
    msr VPIDR_EL2, x0

    //
    // neither EL3 nor EL2 trap floating point or accesses to CPACR
    //
    msr CPTR_EL3, xzr
    msr CPTR_EL2, xzr

    //
```

```
    // SCTLR_ELx may come out of reset with UNKNOWN values so we will
    // set the fields to 0 except, possibly, the endianess field(s).
    // Note that setting SCTLR_EL2 or the EL0 related fields of SCTLR_EL1
    // is not strictly needed, since we are never in EL2 or EL0
    //
#ifdef __ARM_BIG_ENDIAN
    mov x0, #(SCTLR_ELx_EE | SCTLR_EL1_E0E)
#else
    mov x0, #0
#endif
    msr SCTLR_EL3, x0
    msr SCTLR_EL2, x0
    msr SCTLR_EL1, x0

#ifdef CORTEXA
    //
    // Configure ACTLR_EL[23]
    // ----------------------
    //
    // These bits are IMPLEMENTATION DEFINED, so are different for
    // different processors
    //
    // For Cortex-A57, the controls we set are:
    //
    //  Enable lower level access to CPUACTLR_EL1
    //  Enable lower level access to CPUECTLR_EL1
    //  Enable lower level access to L2CTLR_EL1
    //  Enable lower level access to L2ECTLR_EL1
    //  Enable lower level access to L2ACTLR_EL1
    //
    mov x0, #((1 << 0) | \
              (1 << 1) | \
              (1 << 4) | \
              (1 << 5) | \
              (1 << 6))

    msr ACTLR_EL3, x0
    msr ACTLR_EL2, x0

    //
    // configure CPUECTLR_EL1
    //
    // These bits are IMP DEF, so need to be different for different
    // processors
    //
    // SMPEN - bit 6 - Enables the processor to receive cache
    //                 and TLB maintenance operations
    //
    // Note: For Cortex-A57/53 SMPEN should be set before enabling
    //       the caches and MMU, or performing any cache and TLB
    //       maintenance operations.
    //
    //       This register has a defined reset value, so we use a
    //       read-modify-write sequence to set SMPEN
    //
    mrs x0, S3_1_c15_c2_1  // Read EL1 CPU Extended Control Register
    orr x0, x0, #(1 << 6)  // Set the SMPEN bit
    msr S3_1_c15_c2_1, x0  // Write EL1 CPU Extended Control Register

    isb
#endif

    //
    // That is the last of the control settings for now
    //
    // Note: no ISB after all these changes, because registers will not be
    // accessed until after an exception return, which is itself a
    // context synchronization event
    //

    //
```

```
    // Setup some EL3 stack space, ready for calling some subroutines, below.
    //
    ldr x0, =Image$$EL3_STACKS$$ZI$$Limit
    mov sp, x0

    //
    // we need to configure the GIC while still in secure mode, specifically
    // all PPIs and SPIs have to be programmed as Group1 interrupts
    //

    //
    // Before the GIC can be reliably programmed, we need to
    // enable Affinity Routing, as this affects where the configuration
    // registers are (with Affinity Routing enabled, some registers are
    // in the Redistributor, whereas those same registers are in the
    // Distributor with Affinity Routing disabled (that is, when in GICv2
    // compatibility mode).
    //
    mov x0, #(1 << 4) | (1 << 5) // gicdctlr_ARE_S | gicdctlr_ARE_NS
    mov x1, x19
    bl  SyncAREinGICD

    //
    // The Redistributor comes out of reset assuming the processor is
    // asleep - correct that assumption
    //
    mov w0, w19
    bl  WakeupGICR

    //
    // Now we are ready to set security and other initializations
    //
    // This is a per-CPU configuration for these interrupts
    //
    // for the first cluster, CPU number is the redistributor index
    //
    mov w0, w19
    mov w1, #1    // gicigroupr_G1NS
    bl  SetPrivateIntSecurityBlock

    //
    // While we are in the Secure World, set the priority mask low enough
    // for it to be writable in the Non-Secure World
    //
    //mov x0, #16 << 3    // 5 bits of priority in the Secure world
    mov x0, #0xFF  // for Non-Secure interrupts
    msr ICC_PMR_EL1, x0

    //
    // There is more to do to the GIC - call the utility routine to set
    // all SPIs to Group1
    //
    mov w0, #1    // gicigroupr_G1NS
    bl  SetSPISecurityAll

    //
    // Set up EL1 entry point and "dummy" exception return information,
    // then perform exception return to enter EL1
    //
    .global drop_to_el1
drop_to_el1:
    adr x1, el1_entry_aarch64
    msr ELR_EL3, x1
    mov x1, #(AARCH64_SPSR_EL1h | \
              AARCH64_SPSR_F  | \
              AARCH64_SPSR_I  | \
              AARCH64_SPSR_A)
    msr SPSR_EL3, x1
    eret
```

```
// -----------------------------------------------------------
// EL1 - Common start-up code
// -----------------------------------------------------------

    .global el1_entry_aarch64
    .type el1_entry_aarch64, "function"
el1_entry_aarch64:

    //
    // Now we are in EL1, setup the application stack
    //
    ldr x0, =Image$$ARM_LIB_STACK$$ZI$$Limit
    mov sp, x0

    //
    // Enable floating point
    //
    mov x0, #CPACR_EL1_FPEN
    msr CPACR_EL1, x0

    //
    // Invalidate caches and TLBs for all stage 1
    // translations used at EL1
    //
    // Cortex-A processors automatically invalidate their caches on reset
    // (unless suppressed with the DBGL1RSTDISABLE or L2RSTDISABLE pins).
    // It is therefore not necessary for software to invalidate the caches
    // on startup, however, this is done here in case of a warm reset.
    bl  InvalidateUDCaches
    tlbi VMALLE1


    //
    // Set TTBR0 Base address
    //
    // The CPUs share one set of translation tables that are
    // generated by CPU0 at run-time
    //
    // TTBR1_EL1 is not used in this example
    //
    ldr x1, =Image$$TTB0_L1$$ZI$$Base
    msr TTBR0_EL1, x1


    //
    // Set up memory attributes
    //
    // These equate to:
    //
    // 0 -> 0b01000100 = 0x00000044 = Normal, Inner/Outer Non-Cacheable
    // 1 -> 0b11111111 = 0x0000ff00 = Normal, Inner/Outer WriteBack Read/Write
 Allocate
    // 2 -> 0b00000100 = 0x00040000 = Device-nGnRE
    //
    mov  x1, #0xff44
    movk x1, #4, LSL #16    // equiv to: movk x1, #0x0000000000040000
    msr MAIR_EL1, x1


    //
    // Set up TCR_EL1
    //
    // We are using only TTBR0 (EPD1 = 1), and the page table entries:
    //  - are using an 8-bit ASID from TTBR0
    //  - have a 4K granularity (TG0 = 0b00)
    //  - are outer-shareable (SH0 = 0b10)
    //  - are using Inner & Outer WBWA Normal memory ([IO]RGN0 = 0b01)
    //  - map
    //      + 32 bits of VA space (T0SZ = 0x20)
    //      + into a 32-bit PA space (IPS = 0b000)
```

```
//
//     36   32   28   24   20   16   12    8    4    0
//  -----+----+----+----+----+----+----+----+----+----+
//       |    |    |OOII|    |    |    |OOII|    |    |
//    TT |    |    |RRRR|E T |   T|    |RRRR|E T |   T|
//    BB | I I|TTSS|GGGG|P 1 |   1|TTSS|GGGG|P 0 |   0|
//    IIA| P P|GGHH|NNNN|DAS |   S|GGHH|NNNN|D S |   S|
//    10S| S-S|1111|1111|11Z-|---Z|0000|0000|0 Z-|---Z|
//
//    000 0000 0000 0000 1000 0000 0010 0101 0010 0000
//
//                     0x    8    0    2    5    2    0
//
// Note: the ISB is needed to ensure the changes to system
//       context are before the write of SCTLR_EL1.M to enable
//       the MMU. It is likely on a "real" implementation that
//       this setup would work without an ISB, due to the
//       amount of code that gets executed before enabling the
//       MMU, but that would not be architecturally correct.
//
ldr x1, =0x0000000000802520
msr TCR_EL1, x1
isb

//
// Turn on the banked GIC distributor enable,
// ready for individual CPU enables later
//
mov w0, #(1 << 1)  // gicdctlr_EnableGrp1A
bl  EnableGICD

//
// Generate TTBR0 L1
//
// at 4KB granularity, 32-bit VA space, table lookup starts at
// L1, with 1GB regions
//
// we are going to create entries pointing to L2 tables for a
// couple of these 1GB regions, the first of which is the
// RAM on the VE board model - get the table addresses and
// start by emptying out the L1 page tables (4 entries at L1
// for a 4K granularity)
//
// x21 = address of L1 tables
//
ldr x21, =Image$$TTB0_L1$$ZI$$Base
mov x0, x21
mov x1, #(4 << 3)
bl  ZeroBlock

//
// time to start mapping the RAM regions - clear out the
// L2 tables and point to them from the L1 tables
//
// x22 = address of L2 tables, needs to be remembered in case
//       we want to re-use the tables for mapping peripherals
//
ldr x22, =Image$$TTB0_L2_RAM$$ZI$$Base
mov x1, #(512 << 3)
mov x0, x22
bl  ZeroBlock

//
// Get the start address of RAM (the EXEC region) into x4
// and calculate the offset into the L1 table (1GB per region,
// max 4GB)
//
// x23 = L1 table offset, saved for later comparison against
//       peripheral offset
//
ldr x4, =Image$$EXEC$$RO$$Base
```

```
    ubfx x23, x4, #30, #2

    orr x1, x22, #TT_S1_ATTR_PAGE
    str x1, [x21, x23, lsl #3]

    //
    // We have already used the RAM start address in x4 - we now need
    // to get this in terms of an offset into the L2 page tables,
    // where each entry covers 2MB
    //
    ubfx x2, x4, #21, #9

    //
    // TOP_OF_RAM in the scatter file marks the end of the
    // Execute region in RAM: convert the end of this region to an
    // offset too, being careful to round up, then calculate the
    // number of entries to write
    //
    ldr x5, =Image$$TOP_OF_RAM$$ZI$$Base
    sub  x3, x5, #1
    ubfx x3, x3, #21, #9
    add  x3, x3, #1
    sub  x3, x3, x2

    //
    // set x1 to the required page table attributes, then orr
    // in the start address (modulo 2MB)
    //
    // L2 tables in our configuration cover 2MB per entry - map
    // memory as Shared, Normal WBWA (MAIR[1]) with a flat
    // VA->PA translation
    //
    bic x4, x4, #((1 << 21) - 1)
    ldr x1, =(TT_S1_ATTR_BLOCK | \
             (1 << TT_S1_ATTR_MATTR_LSB) | \
              TT_S1_ATTR_NS | \
              TT_S1_ATTR_AP_RW_PL1 | \
              TT_S1_ATTR_SH_INNER | \
              TT_S1_ATTR_AF | \
              TT_S1_ATTR_nG)
    orr x1, x1, x4

    //
    // factor the offset into the page table address and then write
    // the entries
    //
    add x0, x22, x2, lsl #3

loop1:
    subs x3, x3, #1
    str x1, [x0], #8
    add x1, x1, #0x200, LSL #12    // equiv to add x1, x1, #(1 << 21)  // 2MB per
 entry
    bne loop1


    //
    // now mapping the Peripheral regions - clear out the
    // L2 tables and point to them from the L1 tables
    //
    // The assumption here is that all peripherals live within
    // a common 1GB region (that is, that there is a single set of
    // L2 pages for all the peripherals). We only use a UART
    // and the GIC in this example, so the assumption is sound
    //
    // x24 = address of L2 peripheral tables
    //
    ldr x24, =Image$$TTB0_L2_PERIPH$$ZI$$Base

    //
    // get the GICD address into x4 and calculate
```

```
    // the offset into the L1 table
    //
    // x25 = L1 table offset
    //
    ldr x4, =Image$$GICD$$ZI$$Base
    ubfx x25, x4, #30, #2

    //
    // Here is the tricky bit: it is possible that the peripherals are
    // in the same 1GB region as the RAM, in which case we do not need
    // to prime a separate set of L2 page tables, nor add them to the
    // L1 tables
    //
    // if we are going to re-use the TTB0_L2_RAM tables, get their
    // address into x24, which is used later on to write the PTEs
    //
    cmp x25, x23
    csel x24, x22, x24, EQ
    b.eq nol2setup

    //
    // Peripherals are in a separate 1GB region, and so have their own
    // set of L2 tables - clean out the tables and add them to the L1
    // table
    //
    mov x0, x24
    mov x1, #512 << 3
    bl  ZeroBlock

    orr x1, x24, #TT_S1_ATTR_PAGE
    str x1, [x21, x25, lsl #3]

    //
    // there is only going to be a single 2MB region for GICD (in
    // x4) - get this in terms of an offset into the L2 page tables
    //
    // with larger systems, it is possible that the GIC redistributor
    // registers require extra 2MB pages, in which case extra code
    // would be required here
    //
nol2setup:
    ubfx x2, x4, #21, #9

    //
    // set x1 to the required page table attributes, then orr
    // in the start address (modulo 2MB)
    //
    // L2 tables in our configuration cover 2MB per entry - map
    // memory as NS Device-nGnRE (MAIR[2]) with a flat VA->PA
    // translation
    //
    bic x4, x4, #((1 << 21) - 1)  // start address mod 2MB
    ldr x1, =(TT_S1_ATTR_BLOCK | \
              (2 << TT_S1_ATTR_MATTR_LSB) | \
               TT_S1_ATTR_NS | \
               TT_S1_ATTR_AP_RW_PL1 | \
               TT_S1_ATTR_AF | \
               TT_S1_ATTR_nG)
    orr x1, x1, x4

    //
    // only a single L2 entry for this, so no loop as we have for RAM, above
    //
    str x1, [x24, x2, lsl #3]

    //
    // we have CS3_PERIPHERALS that include the UART controller
    //
    // Again, the code is making assumptions - this time that the CS3_PERIPHERALS
    // region uses the same 1GB portion of the address space as the GICD,
    // and thus shares the same set of L2 page tables
```

```
    //
    // Get CS3_PERIPHERALS address into x4 and calculate the offset into the
    // L2 tables
    //
    ldr x4, =Image$$CS3_PERIPHERALS$$ZI$$Base
    ubfx x2, x4, #21, #9

    //
    // set x1 to the required page table attributes, then orr
    // in the start address (modulo 2MB)
    //
    // L2 tables in our configuration cover 2MB per entry - map
    // memory as NS Device-nGnRE (MAIR[2]) with a flat VA->PA
    // translation
    //
    bic x4, x4, #((1 << 21) - 1)  // start address mod 2MB
    ldr x1, =(TT_S1_ATTR_BLOCK | \
              (2 << TT_S1_ATTR_MATTR_LSB) | \
               TT_S1_ATTR_NS | \
               TT_S1_ATTR_AP_RW_PL1 | \
               TT_S1_ATTR_AF | \
               TT_S1_ATTR_nG)
    orr x1, x1, x4

    //
    // only a single L2 entry again - write it
    //
    str x1, [x24, x2, lsl #3]

    //
    // issue a barrier to ensure all table entry writes are complete
    //
    dsb ish

    //
    // Enable the MMU.  Caches will be enabled later, after scatterloading.
    //
    mrs x1, SCTLR_EL1
    orr x1, x1, #SCTLR_ELx_M
    bic x1, x1, #SCTLR_ELx_A // Disable alignment fault checking.  To enable, change
 bic to orr
    msr SCTLR_EL1, x1
    isb

    //
    // Branch to C library init code
    //
    b  __main


// ------------------------------------------------------------

// AArch64 Arm C library startup add-in:

// The Arm Architecture Reference Manual for Armv8-A states:
//
//     Instruction accesses to Non-cacheable Normal memory can be held in
 instruction caches.
//     Correspondingly, the sequence for ensuring that modifications to instructions
 are available
//     for execution must include invalidation of the modified locations from the
 instruction cache,
//     even if the instructions are held in Normal Non-cacheable memory.
//     This includes cases where the instruction cache is disabled.
//
// To invalidate the AArch64 instruction cache after scatter-loading and before
 initialization of the stack and heap,
// it is necessary for the user to:
//
// * Implement instruction cache invalidation code in _platform_pre_stackheap_init.
```

```
// * Ensure all code on the path from the program entry up to and including
 _platform_pre_stackheap_init is located in a root region.

    .global _platform_pre_stackheap_init
    .type _platform_pre_stackheap_init, "function"
    .cfi_startproc
_platform_pre_stackheap_init:
    dsb ish     // ensure all previous stores have completed before invalidating
    ic  ialluis // I cache invalidate all inner shareable to PoU (which includes
 secondary cores)
    dsb ish     // ensure completion on inner shareable domain   (which includes
 secondary cores)
    isb

    // Scatter-loading is complete, so enable the caches here, so that the C-
library's mutex initialization later will work
    mrs x1, SCTLR_EL1
    orr x1, x1, #SCTLR_ELx_C
    orr x1, x1, #SCTLR_ELx_I
    msr SCTLR_EL1, x1
    isb

    ret
    .cfi_endproc
```

Create the file `v8_aarch64.s` containing the following code:

```
// ----------------------------------------------------------------
// Armv8-A AArch64 - Common helper functions
//
// Copyright (c) 2012-2020 Arm Limited (or its affiliates). All rights reserved.
// Use, modification and redistribution of this file is subject to your possession
 of a
// valid End User License Agreement for the Arm Product of which these examples are
 part of
// and your compliance with all applicable terms and conditions of such licence
 agreement.
// ----------------------------------------------------------------

#include "v8_system.h"

    .text
    .cfi_sections .debug_frame  // put stack frame info into .debug_frame instead
 of .eh_frame

    .global EnableCachesEL1
    .global DisableCachesEL1
    .global InvalidateUDCaches
    .global GetMIDR
    .global GetMPIDR
    .global GetCPUID

// ----------------------------------------------------------------

//
// void EnableCachesEL1(void)
//
//    enable Instruction and Data caches
//
    .type EnableCachesEL1, "function"
    .cfi_startproc
EnableCachesEL1:

    mrs x0, SCTLR_EL1
    orr x0, x0, #SCTLR_ELx_I
    orr x0, x0, #SCTLR_ELx_C
    msr SCTLR_EL1, x0
```

```
    isb
    ret
    .cfi_endproc


// ----------------------------------------------------------------

    .type DisableCachesEL1, "function"
    .cfi_startproc
DisableCachesEL1:

    mrs x0, SCTLR_EL1
    bic x0, x0, #SCTLR_ELx_I
    bic x0, x0, #SCTLR_ELx_C
    msr SCTLR_EL1, x0

    isb
    ret
    .cfi_endproc


// ----------------------------------------------------------------

//
// void InvalidateUDCaches(void)
//
//    Invalidate data and unified caches
//
    .type InvalidateUDCaches, "function"
    .cfi_startproc
InvalidateUDCaches:
    // From the Armv8-A Architecture Reference Manual

    dmb  ish                  // ensure all prior inner-shareable accesses have
 been observed

    mrs  x0, CLIDR_EL1
    and  w3, w0, #0x07000000  // get 2 x level of coherence
    lsr  w3, w3, #23
    cbz  w3, finished
    mov  w10, #0              // w10 = 2 x cache level
    mov  w8, #1               // w8 = constant 0b1
loop_level:
    add  w2, w10, w10, lsr #1 // calculate 3 x cache level
    lsr  w1, w0, w2           // extract 3-bit cache type for this level
    and  w1, w1, #0x7
    cmp  w1, #2
    b.lt next_level           // no data or unified cache at this level
    msr  CSSELR_EL1, x10      // select this cache level
    isb                       // synchronize change of csselr
    mrs  x1, CCSIDR_EL1       // read ccsidr
    and  w2, w1, #7           // w2 = log2(linelen)-4
    add  w2, w2, #4           // w2 = log2(linelen)
    ubfx w4, w1, #3, #10      // w4 = max way number, right aligned
    clz  w5, w4               // w5 = 32-log2(ways), bit position of way in dc
 operand
    lsl  w9, w4, w5           // w9 = max way number, aligned to position in dc
 operand
    lsl  w16, w8, w5          // w16 = amount to decrement way number per iteration
loop_way:
    ubfx w7, w1, #13, #15     // w7 = max set number, right aligned
    lsl  w7, w7, w2           // w7 = max set number, aligned to position in dc
 operand
    lsl  w17, w8, w2          // w17 = amount to decrement set number per iteration
loop_set:
    orr  w11, w10, w9         // w11 = combine way number and cache number ...
    orr  w11, w11, w7         // ... and set number for dc operand
    dc   isw, x11             // do data cache invalidate by set and way
    subs w7, w7, w17          // decrement set number
    b.ge loop_set
    subs x9, x9, x16          // decrement way number
```

```
    b.ge loop_way
next_level:
    add  w10, w10, #2          // increment 2 x cache level
    cmp  w3, w10
    b.gt loop_level
    dsb  sy                    // ensure completion of previous cache maintenance
 operation
    isb
finished:
    ret
    .cfi_endproc


// ------------------------------------------------------------

//
// ID Register functions
//

    .type GetMIDR, "function"
    .cfi_startproc
GetMIDR:

    mrs x0, MIDR_EL1
    ret
    .cfi_endproc


    .type GetMPIDR, "function"
    .cfi_startproc
GetMPIDR:

    mrs x0, MPIDR_EL1
    ret
    .cfi_endproc


    .type GetCPUID, "function"
    .cfi_startproc
GetCPUID:

    mrs x0, MIDR_EL1
    ubfx    x0, x0, #4, #12 // extract PartNum
    cmp x0, #0xD0F          // AEMv8-A FVP
    b.eq Others
    cmp x0, #0xD05          // Cortex-A55
    b.eq DynamIQ
    cmp x0, #0xD06          // Cortex-A65
    b.eq DynamIQ
    cmp x0, #0xD0A          // Cortex-A75 or higher
    b.pl DynamIQ
    b Others
DynamIQ:
    mrs x0, MPIDR_EL1
    ubfx    x0, x0, #MPIDR_EL1_AFF1_LSB, #MPIDR_EL1_AFF_WIDTH
    ret

Others:
    mrs x0, MPIDR_EL1
    ubfx    x1, x0, #MPIDR_EL1_AFF0_LSB, #MPIDR_EL1_AFF_WIDTH
    ubfx    x2, x0, #MPIDR_EL1_AFF1_LSB, #MPIDR_EL1_AFF_WIDTH
    add     x0, x1, x2, LSL #2
    ret
    .cfi_endproc
```

Create the file `v8_mmu.h` containing the following code:

```
//
```

```
// Defines for v8 Memory Model
//
// Copyright (c) 2012-2019 Arm Limited (or its affiliates). All rights reserved.
// Use, modification and redistribution of this file is subject to your possession
 of a
// valid End User License Agreement for the Arm Product of which these examples are
 part of
// and your compliance with all applicable terms and conditions of such licence
 agreement.
//

#ifndef V8_MMU_H
#define V8_MMU_H

//
// Translation Control Register fields
//
// RGN field encodings
//
#define TCR_RGN_NC   0b00
#define TCR_RGN_WBWA 0b01
#define TCR_RGN_WT   0b10
#define TCR_RGN_WBRA 0b11

//
// Shareability encodings
//
#define TCR_SHARE_NONE  0b00
#define TCR_SHARE_OUTER 0b10
#define TCR_SHARE_INNER 0b11

//
// Granule size encodings
//
#define TCR_GRANULE_4K  0b00
#define TCR_GRANULE_64K 0b01
#define TCR_GRANULE_16K 0b10

//
// Physical Address sizes
//
#define TCR_SIZE_4G   0b000
#define TCR_SIZE_64G  0b001
#define TCR_SIZE_1T   0b010
#define TCR_SIZE_4T   0b011
#define TCR_SIZE_16T  0b100
#define TCR_SIZE_256T 0b101

//
// Translation Control Register fields
//
#define TCR_EL1_T0SZ_SHIFT   0
#define TCR_EL1_EPD0  (1 << 7)
#define TCR_EL1_IRGN0_SHIFT  8
#define TCR_EL1_ORGN0_SHIFT 10
#define TCR_EL1_SH0_SHIFT   12
#define TCR_EL1_TG0_SHIFT   14

#define TCR_EL1_T1SZ_SHIFT  16
#define TCR_EL1_A1   (1 << 22)
#define TCR_EL1_EPD1 (1 << 23)
#define TCR_EL1_IRGN1_SHIFT 24
#define TCR_EL1_ORGN1_SHIFT 26
#define TCR_EL1_SH1_SHIFT   28
#define TCR_EL1_TG1_SHIFT   30
#define TCR_EL1_IPS_SHIFT   32
#define TCR_EL1_AS   (1 << 36)
#define TCR_EL1_TBI0 (1 << 37)
#define TCR_EL1_TBI1 (1 << 38)

//
```

```
// Stage 1 Translation Table descriptor fields
//
#define TT_S1_ATTR_FAULT (0b00 << 0)
#define TT_S1_ATTR_BLOCK (0b01 << 0) // Level 1/2
#define TT_S1_ATTR_TABLE (0b11 << 0) // Level 0/1/2
#define TT_S1_ATTR_PAGE  (0b11 << 0) // Level 3

#define TT_S1_ATTR_MATTR_LSB 2

#define TT_S1_ATTR_NS (1 << 5)

#define TT_S1_ATTR_AP_RW_PL1 (0b00 << 6)
#define TT_S1_ATTR_AP_RW_ANY (0b01 << 6)
#define TT_S1_ATTR_AP_RO_PL1 (0b10 << 6)
#define TT_S1_ATTR_AP_RO_ANY (0b11 << 6)

#define TT_S1_ATTR_SH_NONE   (0b00 << 8)
#define TT_S1_ATTR_SH_OUTER  (0b10 << 8)
#define TT_S1_ATTR_SH_INNER  (0b11 << 8)

#define TT_S1_ATTR_AF (1 << 10)
#define TT_S1_ATTR_nG (1 << 11)

// OA bits [15:12] - If Armv8.2-LPA is implemented, bits[15:12] are bits[51:48]
// and bits[47:16] are bits[47:16] of the output address for a page of memory

#define TT_S1_ATTR_nT (1 << 16) // Present if Armv8.4-TTRem is implemented,
 otherwise RES0

#define TT_S1_ATTR_DBM (1 << 51) // Present if Armv8.1-TTHM is implemented,
 otherwise RES0

#define TT_S1_ATTR_CONTIG (1 << 52)
#define TT_S1_ATTR_PXN    (1 << 53)
#define TT_S1_ATTR_UXN    (1 << 54)

// PBHA bits[62:59] - If Armv8.2-TTPBHA is implemented, hardware can use these bits
// for IMPLEMENTATIONDEFINED purposes, otherwise IGNORED

#define TT_S1_MAIR_DEV_nGnRnE 0b00000000
#define TT_S1_MAIR_DEV_nGnRE  0b00000100
#define TT_S1_MAIR_DEV_nGRE   0b00001000
#define TT_S1_MAIR_DEV_GRE    0b00001100

//
// Inner and Outer Normal memory attributes use the same bit patterns
// Outer attributes just need to be shifted up
//
#define TT_S1_MAIR_OUTER_SHIFT 4

#define TT_S1_MAIR_WT_TRANS_RA  0b0010

#define TT_S1_MAIR_WB_TRANS_RA  0b0110
#define TT_S1_MAIR_WB_TRANS_RWA 0b0111

#define TT_S1_MAIR_WT_RA  0b1010

#define TT_S1_MAIR_WB_RA  0b1110
#define TT_S1_MAIR_WB_RWA 0b1111

#endif // V8_MMU_H
```

Create the file `v8_system.h` containing the following code:

```
//
// Defines for v8 System Registers
//
// Copyright (c) 2012-2016 Arm Limited (or its affiliates). All rights reserved.
```

```
// Use, modification and redistribution of this file is subject to your possession
 of a
// valid End User License Agreement for the Arm Product of which these examples are
 part of
// and your compliance with all applicable terms and conditions of such licence
 agreement.
//

#ifndef V8_SYSTEM_H
#define V8_SYSTEM_H

//
// AArch64 SPSR
//
#define AARCH64_SPSR_EL3h 0b1101
#define AARCH64_SPSR_EL3t 0b1100
#define AARCH64_SPSR_EL2h 0b1001
#define AARCH64_SPSR_EL2t 0b1000
#define AARCH64_SPSR_EL1h 0b0101
#define AARCH64_SPSR_EL1t 0b0100
#define AARCH64_SPSR_EL0t 0b0000
#define AARCH64_SPSR_RW (1 << 4)
#define AARCH64_SPSR_F  (1 << 6)
#define AARCH64_SPSR_I  (1 << 7)
#define AARCH64_SPSR_A  (1 << 8)
#define AARCH64_SPSR_D  (1 << 9)
#define AARCH64_SPSR_IL (1 << 20)
#define AARCH64_SPSR_SS (1 << 21)
#define AARCH64_SPSR_V  (1 << 28)
#define AARCH64_SPSR_C  (1 << 29)
#define AARCH64_SPSR_Z  (1 << 30)
#define AARCH64_SPSR_N  (1 << 31)

//
// Multiprocessor Affinity Register
//
#define MPIDR_EL1_AFF3_LSB 32
#define MPIDR_EL1_U  (1 << 30)
#define MPIDR_EL1_MT (1 << 24)
#define MPIDR_EL1_AFF2_LSB 16
#define MPIDR_EL1_AFF1_LSB  8
#define MPIDR_EL1_AFF0_LSB  0
#define MPIDR_EL1_AFF_WIDTH 8

//
// Data Cache Zero ID Register
//
#define DCZID_EL0_BS_LSB   0
#define DCZID_EL0_BS_WIDTH 4
#define DCZID_EL0_DZP_LSB  5
#define DCZID_EL0_DZP (1 << 5)

//
// System Control Register
//
#define SCTLR_EL1_UCI     (1 << 26)
#define SCTLR_ELx_EE      (1 << 25)
#define SCTLR_EL1_E0E     (1 << 24)
#define SCTLR_ELx_WXN     (1 << 19)
#define SCTLR_EL1_nTWE    (1 << 18)
#define SCTLR_EL1_nTWI    (1 << 16)
#define SCTLR_EL1_UCT     (1 << 15)
#define SCTLR_EL1_DZE     (1 << 14)
#define SCTLR_ELx_I       (1 << 12)
#define SCTLR_EL1_UMA     (1 << 9)
#define SCTLR_EL1_SED     (1 << 8)
#define SCTLR_EL1_ITD     (1 << 7)
#define SCTLR_EL1_THEE    (1 << 6)
#define SCTLR_EL1_CP15BEN (1 << 5)
#define SCTLR_EL1_SA0     (1 << 4)
#define SCTLR_ELx_SA      (1 << 3)
```

```
#define SCTLR_ELx_C       (1 << 2)
#define SCTLR_ELx_A       (1 << 1)
#define SCTLR_ELx_M       (1 << 0)

//
// Architectural Feature Access Control Register
//
#define CPACR_EL1_TTA     (1 << 28)
#define CPACR_EL1_FPEN    (3 << 20)

//
// Architectural Feature Trap Register
//
#define CPTR_ELx_TCPAC (1 << 31)
#define CPTR_ELx_TTA   (1 << 20)
#define CPTR_ELx_TFP   (1 << 10)

//
// Secure Configuration Register
//
#define SCR_EL3_TWE  (1 << 13)
#define SCR_EL3_TWI  (1 << 12)
#define SCR_EL3_ST   (1 << 11)
#define SCR_EL3_RW   (1 << 10)
#define SCR_EL3_SIF  (1 << 9)
#define SCR_EL3_HCE  (1 << 8)
#define SCR_EL3_SMD  (1 << 7)
#define SCR_EL3_EA   (1 << 3)
#define SCR_EL3_FIQ  (1 << 2)
#define SCR_EL3_IRQ  (1 << 1)
#define SCR_EL3_NS   (1 << 0)

//
// Hypervisor Configuration Register
//
#define HCR_EL2_ID   (1 << 33)
#define HCR_EL2_CD   (1 << 32)
#define HCR_EL2_RW   (1 << 31)
#define HCR_EL2_TRVM (1 << 30)
#define HCR_EL2_HVC  (1 << 29)
#define HCR_EL2_TDZ  (1 << 28)

#endif // V8_SYSTEM_H
```

Create the file `v8_utils.s` containing the following code:

```
//
// Simple utility routines for baremetal v8 code
//
// Copyright (c) 2013-2017 Arm Limited (or its affiliates). All rights reserved.
// Use, modification and redistribution of this file is subject to your possession
 of a
// valid End User License Agreement for the Arm Product of which these examples are
 part of
// and your compliance with all applicable terms and conditions of such licence
 agreement.
//

#include "v8_system.h"

    .text
    .cfi_sections .debug_frame  // put stack frame info into .debug_frame instead
 of .eh_frame

//
// void *ZeroBlock(void *blockPtr, unsigned int nBytes)
//
// Zero fill a block of memory
```

```
//   Fill memory pages or similar structures with zeros.
//   The byte count must be a multiple of the block fill size (16 bytes)
//
//  Inputs:
//    blockPtr - base address of block to fill
//    nBytes - block size, in bytes
//
//  Returns:
//    pointer to just filled block, NULL if nBytes is
//  incompatible with block fill size
//
    .global ZeroBlock
    .type ZeroBlock, "function"
    .cfi_startproc
ZeroBlock:

    //
    // we fill data by steam, 16 bytes at a time: check that
    // blocksize is a multiple of that
    //
    ubfx x2, x1, #0, #4
    cbnz x2, incompatible

    //
    // we already have one register full of zeros, get another
    //
    mov x3, x2

    //
    // OK, set temporary pointer and away we go
    //
    add x0, x0, x1

loop0:
    subs x1, x1, #16
    stp  x2, x3, [x0, #-16]!
    b.ne loop0

    //
    // that's all - x0 will be back to its start value
    //
    ret

    //
    // parameters are incompatible with block size - return
    // an indication that this is so
    //
incompatible:
    mov x0,#0
    ret
    .cfi_endproc
```

Create the file `vectors.s` containing the following code:

```
// -------------------------------------------------------------
// Armv8-A Vector tables
//
// Copyright (c) 2014-2016 Arm Limited (or its affiliates). All rights reserved.
// Use, modification and redistribution of this file is subject to your possession
 of a
// valid End User License Agreement for the Arm Product of which these examples are
 part of
// and your compliance with all applicable terms and conditions of such licence
 agreement.
// -------------------------------------------------------------


    .global el1_vectors
```

```
    .global el2_vectors
    .global el3_vectors
    .global c0sync1
    .global irqHandler
    .global fiqHandler
    .global irqFirstLevelHandler
    .global fiqFirstLevelHandler

    .section  EL1VECTORS, "ax"
    .align 11

//
// Current EL with SP0
//
el1_vectors:
c0sync1: B c0sync1

    .balign 0x80
c0irq1: B irqFirstLevelHandler

    .balign 0x80
c0fiq1: B fiqFirstLevelHandler

    .balign 0x80
c0serr1: B c0serr1

//
// Current EL with SPx
//
    .balign 0x80
cxsync1: B cxsync1

    .balign 0x80
cxirq1: B irqFirstLevelHandler

    .balign 0x80
cxfiq1: B fiqFirstLevelHandler

    .balign 0x80
cxserr1: B cxserr1

//
// Lower EL using AArch64
//
    .balign 0x80
l64sync1: B l64sync1

    .balign 0x80
l64irq1: B irqFirstLevelHandler

    .balign 0x80
l64fiq1: B fiqFirstLevelHandler

    .balign 0x80
l64serr1: B l64serr1

//
// Lower EL using AArch32
//
    .balign 0x80
l32sync1: B l32sync1

    .balign 0x80
l32irq1: B irqFirstLevelHandler

    .balign 0x80
l32fiq1: B fiqFirstLevelHandler

    .balign 0x80
l32serr1: B l32serr1
```

```
//-----------------------------------------------------------------

    .section  EL2VECTORS, "ax"
    .align 11

//
// Current EL with SP0
//
el2_vectors:
c0sync2: B c0sync2

    .balign 0x80
c0irq2: B irqFirstLevelHandler

    .balign 0x80
c0fiq2: B fiqFirstLevelHandler

    .balign 0x80
c0serr2: B c0serr2

//
// Current EL with SPx
//
    .balign 0x80
cxsync2: B cxsync2

    .balign 0x80
cxirq2: B irqFirstLevelHandler

    .balign 0x80
cxfiq2: B fiqFirstLevelHandler

    .balign 0x80
cxserr2: B cxserr2

//
// Lower EL using AArch64
//
    .balign 0x80
l64sync2: B l64sync2

    .balign 0x80
l64irq2: B irqFirstLevelHandler

    .balign 0x80
l64fiq2: B fiqFirstLevelHandler

    .balign 0x80
l64serr2: B l64serr2

//
// Lower EL using AArch32
//
    .balign 0x80
l32sync2: B l32sync2

    .balign 0x80
l32irq2: B irqFirstLevelHandler

    .balign 0x80
l32fiq2: B fiqFirstLevelHandler

    .balign 0x80
l32serr2: B l32serr2

//-----------------------------------------------------------------

    .section  EL3VECTORS, "ax"
    .align 11

//
```

```
// Current EL with SP0
//
el3_vectors:
c0sync3: B c0sync3

    .balign 0x80
c0irq3: B irqFirstLevelHandler

    .balign 0x80
c0fiq3: B fiqFirstLevelHandler

    .balign 0x80
c0serr3: B c0serr3

//
// Current EL with SPx
//
    .balign 0x80
cxsync3: B cxsync3

    .balign 0x80
cxirq3: B irqFirstLevelHandler

    .balign 0x80
cxfiq3: B fiqFirstLevelHandler

    .balign 0x80
cxserr3: B cxserr3

//
// Lower EL using AArch64
//
    .balign 0x80
l64sync3: B l64sync3

    .balign 0x80
l64irq3: B irqFirstLevelHandler

    .balign 0x80
l64fiq3: B fiqFirstLevelHandler

    .balign 0x80
l64serr3: B l64serr3

//
// Lower EL using AArch32
//
    .balign 0x80
l32sync3: B l32sync3

    .balign 0x80
l32irq3: B irqFirstLevelHandler

    .balign 0x80
l32fiq3: B fiqFirstLevelHandler

    .balign 0x80
l32serr3: B l32serr3


    .section InterruptHandlers, "ax"
    .balign 4

    .type irqFirstLevelHandler, "function"
irqFirstLevelHandler:
  STP      x29, x30, [sp, #-16]!
  STP      x18, x19, [sp, #-16]!
  STP      x16, x17, [sp, #-16]!
  STP      x14, x15, [sp, #-16]!
  STP      x12, x13, [sp, #-16]!
  STP      x10, x11, [sp, #-16]!
```

```
  STP       x8, x9, [sp, #-16]!
  STP       x6, x7, [sp, #-16]!
  STP       x4, x5, [sp, #-16]!
  STP       x2, x3, [sp, #-16]!
  STP       x0, x1, [sp, #-16]!

  BL        irqHandler

  LDP       x0, x1, [sp], #16
  LDP       x2, x3, [sp], #16
  LDP       x4, x5, [sp], #16
  LDP       x6, x7, [sp], #16
  LDP       x8, x9, [sp], #16
  LDP       x10, x11, [sp], #16
  LDP       x12, x13, [sp], #16
  LDP       x14, x15, [sp], #16
  LDP       x16, x17, [sp], #16
  LDP       x18, x19, [sp], #16
  LDP       x29, x30, [sp], #16
  ERET

    .type fiqFirstLevelHandler, "function"
fiqFirstLevelHandler:
  STP       x29, x30, [sp, #-16]!
  STP       x18, x19, [sp, #-16]!
  STP       x16, x17, [sp, #-16]!
  STP       x14, x15, [sp, #-16]!
  STP       x12, x13, [sp, #-16]!
  STP       x10, x11, [sp, #-16]!
  STP       x8, x9, [sp, #-16]!
  STP       x6, x7, [sp, #-16]!
  STP       x4, x5, [sp, #-16]!
  STP       x2, x3, [sp, #-16]!
  STP       x0, x1, [sp, #-16]!

  BL        fiqHandler

  LDP       x0, x1, [sp], #16
  LDP       x2, x3, [sp], #16
  LDP       x4, x5, [sp], #16
  LDP       x6, x7, [sp], #16
  LDP       x8, x9, [sp], #16
  LDP       x10, x11, [sp], #16
  LDP       x12, x13, [sp], #16
  LDP       x14, x15, [sp], #16
  LDP       x16, x17, [sp], #16
  LDP       x18, x19, [sp], #16
  LDP       x29, x30, [sp], #16
  ERET
```

14.6 C source files for the AArch64 TLS local-exec static linking example

Create the C source files for the AArch64 TLS local-exec static linking example.

List of C source files for the example

- `GICv3.h`
- `GICv3_gicc.h`
- `GICv3_gicd.c`

- GICv3_gicr.c
- main.c
- retarget.c
- sp804_timer.c
- sp804_timer.h
- timer_interrupts.c
- uart.c
- uart.h
- v8_aarch64.h

Contents of the C source files for the example

Create the file `GICv3.h` containing the following code:

```
/*
 * GICv3.h - data types and function prototypes for GICv3 utility routines
 *
 * Copyright (c) 2014-2017 Arm Limited (or its affiliates). All rights reserved.
 * Use, modification and redistribution of this file is subject to your possession
 of a
 * valid End User License Agreement for the Arm Product of which these examples are
 part of
 * and your compliance with all applicable terms and conditions of such licence
 agreement.
 */
#ifndef GICV3_h
#define GICV3_h

#include <stdint.h>

/*
 * extra flags for GICD enable
 */
typedef enum
{
    gicdctlr_EnableGrp0 = (1 << 0),
    gicdctlr_EnableGrp1NS = (1 << 1),
    gicdctlr_EnableGrp1A = (1 << 1),
    gicdctlr_EnableGrp1S = (1 << 2),
    gicdctlr_EnableAll = (1 << 2) | (1 << 1) | (1 << 0),
    gicdctlr_ARE_S = (1 << 4), /* Enable Secure state affinity routing */
    gicdctlr_ARE_NS = (1 << 5), /* Enable Non-Secure state affinity routing */
    gicdctlr_DS = (1 << 6), /* Disable Security support */
    gicdctlr_E1NWF = (1 << 7) /* Enable "1-of-N" wakeup model */
} GICDCTLRFlags_t;

/*
 * modes for SPI routing
 */
typedef enum
{
    gicdirouter_ModeSpecific = 0,
    gicdirouter_ModeAny = (1 << 31)
} GICDIROUTERBits_t;

typedef enum
{
    gicdicfgr_Level = 0,
    gicdicfgr_Edge = (1 << 1)
} GICDICFGRBits_t;
```

```
typedef enum
{
    gicigroupr_G0S = 0,
    gicigroupr_G1NS = (1 << 0),
    gicigroupr_G1S = (1 << 2)
} GICIGROUPRBits_t;

typedef enum
{
    gicrwaker_ProcessorSleep = (1 << 1),
    gicrwaker_ChildrenAsleep = (1 << 2)
} GICRWAKERBits_t;

/*****************************************************************/

/*
 * Utility macros & functions
 */
#define RANGE_LIMIT(x)  ((sizeof(x) / sizeof((x)[0])) - 1)

static inline uint64_t gicv3PackAffinity(uint32_t aff3, uint32_t aff2,
      uint32_t aff1, uint32_t aff0)
{
    /*
     * only need to cast aff3 to get type promotion for all affinities
     */
    return ((((uint64_t)aff3 & 0xff) << 32) |
     ((aff2 & 0xff) << 16) |
     ((aff1 & 0xff) << 8) | aff0);
}

/*****************************************************************/

/*
 * GIC Distributor Function Prototypes
 */

/*
 * ConfigGICD - configure GIC Distributor prior to enabling it
 *
 * Inputs:
 *
 *  control - control flags
 *
 * Returns:
 *
 *  <nothing>
 *
 * NOTE:
 *
 *  ConfigGICD() will set an absolute flags value, whereas
 *  {En,Dis}ableGICD() will only {set,clear} the flag bits
 *  passed as a parameter
 */
void ConfigGICD(GICDCTLRFlags_t flags);

/*
 * EnableGICD - top-level enable for GIC Distributor
 *
 * Inputs:
 *
 *  flags - new control flags to set
 *
 * Returns:
 *
 *  <nothing>
 *
 * NOTE:
 *
 *  ConfigGICD() will set an absolute flags value, whereas
```

```
 *  {En,Dis}ableGICD() will only {set,clear} the flag bits
 *  passed as a parameter
 */
void EnableGICD(GICDCTLRFlags_t flags);

/*
 * DisableGICD - top-level disable for GIC Distributor
 *
 * Inputs
 *
 *  flags - control flags to clear
 *
 * Returns
 *
 *  <nothing>
 *
 * NOTE:
 *
 *  ConfigGICD() will set an absolute flags value, whereas
 *  {En,Dis}ableGICD() will only {set,clear} the flag bits
 *  passed as a parameter
 */
void DisableGICD(GICDCTLRFlags_t flags);

/*
 * SyncAREinGICD - synchronise GICD Address Routing Enable bits
 *
 * Inputs
 *
 *  flags - absolute flag bits to set in GIC Distributor
 *
 *  dosync - flag whether to wait for ARE bits to match passed
 *      flag field (dosync = true), or whether to set absolute
 *      flag bits (dosync = false)
 *
 * Returns
 *
 *  <nothing>
 *
 * NOTE:
 *
 * This function is used to resolve a race in an MP system whereby secondary
 * CPUs cannot reliably program all Redistributor registers until the
 * primary CPU has enabled Address Routing. The primary CPU will call this
 * function with dosync = false, while the secondaries will call it with
 * dosync = true.
 */
void SyncAREinGICD(GICDCTLRFlags_t flags, uint32_t dosync);

/*
 * EnableSPI - enable a specific shared peripheral interrupt
 *
 * Inputs:
 *
 *  id - which interrupt to enable
 *
 * Returns:
 *
 *  <nothing>
 */
void EnableSPI(uint32_t id);

/*
 * DisableSPI - disable a specific shared peripheral interrupt
 *
 * Inputs:
 *
 *  id - which interrupt to disable
 *
 * Returns:
 *
```

```
 *  <nothing>
 */
void DisableSPI(uint32_t id);

/*
 * SetSPIPriority - configure the priority for a shared peripheral interrupt
 *
 * Inputs:
 *
 *  id - interrupt identifier
 *
 *  priority - 8-bit priority to program (see note below)
 *
 * Returns:
 *
 *  <nothing>
 *
 * Note:
 *
 *  The GICv3 architecture makes this function sensitive to the Security
 *  context in terms of what effect it has on the programmed priority: no
 *  attempt is made to adjust for the reduced priority range available
 *  when making Non-Secure accesses to the GIC
 */
void SetSPIPriority(uint32_t id, uint32_t priority);

/*
 * GetSPIPriority - determine the priority for a shared peripheral interrupt
 *
 * Inputs:
 *
 *  id - interrupt identifier
 *
 * Returns:
 *
 *  interrupt priority in the range 0 - 0xff
 */
uint32_t GetSPIPriority(uint32_t id);

/*
 * SetSPIRoute - specify interrupt routing when gicdctlr_ARE is enabled
 *
 * Inputs:
 *
 *  id - interrupt identifier
 *
 *  affinity - prepacked "dotted quad" affinity routing. NOTE: use the
 *      gicv3PackAffinity() helper routine to generate this input
 *
 *  mode - select routing mode (specific affinity, or any recipient)
 *
 * Returns:
 *
 *  <nothing>
 */
void SetSPIRoute(uint32_t id, uint64_t affinity, GICDIROUTERBits_t mode);

/*
 * GetSPIRoute - read ARE-enabled interrupt routing information
 *
 * Inputs:
 *
 *  id - interrupt identifier
 *
 * Returns:
 *
 *  routing configuration
 */
uint64_t GetSPIRoute(uint32_t id);

/*
```

```
 * SetSPITarget - configure the set of processor targets for an interrupt
 *
 * Inputs
 *
 *  id - interrupt identifier
 *
 *  target - 8-bit target bitmap
 *
 * Returns
 *
 *  <nothing>
 */
void SetSPITarget(uint32_t id, uint32_t target);

/*
 * GetSPITarget - read the set of processor targets for an interrupt
 *
 * Inputs
 *
 *  id - interrupt identifier
 *
 * Returns
 *
 *  8-bit target bitmap
 */
uint32_t GetSPITarget(uint32_t id);

/*
 * ConfigureSPI - setup an interrupt as edge- or level-triggered
 *
 * Inputs
 *
 *   id - interrupt identifier
 *
 *   config - desired configuration
 *
 * Returns
 *
 *   <nothing>
 */
void ConfigureSPI(uint32_t id, GICDICFGRBits_t config);

/*
 * SetSPIPending - mark an interrupt as pending
 *
 * Inputs
 *
 *  id - interrupt identifier
 *
 * Returns
 *
 *   <nothing>
 */
void SetSPIPending(uint32_t id);

/*
 * ClearSPIPending - mark an interrupt as not pending
 *
 * Inputs
 *
 *  id - interrupt identifier
 *
 * Returns
 *
 *   <nothing>
 */
void ClearSPIPending(uint32_t id);

/*
 * GetSPIPending - query whether an interrupt is pending
 *
```

```
 * Inputs
 *
 *  id - interrupt identifier
 *
 * Returns
 *
 *   pending status
 */
uint32_t GetSPIPending(uint32_t id);

/*
 * SetSPISecurity - mark a shared peripheral interrupt as
 *     security <group>
 *
 * Inputs
 *
 *  id - which interrupt to mark
 *
 *  group - the group for the interrupt
 *
 * Returns
 *
 *  <nothing>
 */
void SetSPISecurity(uint32_t id, GICIGROUPRBits_t group);

/*
 * SetSPISecurityBlock - mark a block of 32 shared peripheral
 *     interrupts as security <group>
 *
 * Inputs:
 *
 *  block - which block to mark (for example, 1 = Ints 32-63)
 *
 *  group - the group for the interrupts
 *
 * Returns:
 *
 *  <nothing>
 */
void SetSPISecurityBlock(uint32_t block, GICIGROUPRBits_t group);

/*
 * SetSPISecurityAll - mark all shared peripheral interrupts
 *     as security <group>
 *
 * Inputs:
 *
 *  group - the group for the interrupts
 *
 * Returns:
 *
 *  <nothing>
 */
void SetSPISecurityAll(GICIGROUPRBits_t group);

/*********************************************************************/

/*
 * GIC Re-Distributor Function Prototypes
 *
 * The model for calling Redistributor functions is that, rather than
 * identifying the target redistributor with every function call, the
 * SelectRedistributor() function is used to identify which redistributor
 * is to be used for all functions until a different redistributor is
 * explicitly selected
 */

/*
 * WakeupGICR - wake up a Redistributor
 *
```

```
 * Inputs:
 *
 *  gicr - which Redistributor to wakeup
 *
 * Returns:
 *
 *  <nothing>
 */
void WakeupGICR(uint32_t gicr);

/*
 * EnablePrivateInt - enable a private (SGI/PPI) interrupt
 *
 * Inputs:
 *
 *  gicr - which Redistributor to program
 *
 *  id - which interrupt to enable
 *
 * Returns:
 *
 *  <nothing>
 */
void EnablePrivateInt(uint32_t gicr, uint32_t id);

/*
 * DisablePrivateInt - disable a private (SGI/PPI) interrupt
 *
 * Inputs:
 *
 *  gicr - which Redistributor to program
 *
 *  id - which interrupt to disable
 *
 * Returns:
 *
 *  <nothing>
 */
void DisablePrivateInt(uint32_t gicr, uint32_t id);

/*
 * SetPrivateIntPriority - configure the priority for a private
 *     (SGI/PPI) interrupt
 *
 * Inputs:
 *
 *  gicr - which Redistributor to program
 *
 *  id - interrupt identifier
 *
 *  priority - 8-bit priority to program (see note below)
 *
 * Returns:
 *
 *  <nothing>
 *
 * Note:
 *
 *  The GICv3 architecture makes this function sensitive to the Security
 *  context in terms of what effect it has on the programmed priority: no
 *  attempt is made to adjust for the reduced priority range available
 *  when making Non-Secure accesses to the GIC
 */
void SetPrivateIntPriority(uint32_t gicr, uint32_t id, uint32_t priority);

/*
 * GetPrivateIntPriority - configure the priority for a private
 *     (SGI/PPI) interrupt
 *
 * Inputs:
 *
```

```
 *  gicr - which Redistributor to program
 *
 *  id - interrupt identifier
 *
 * Returns:
 *
 *   Int priority
 */
uint32_t GetPrivateIntPriority(uint32_t gicr, uint32_t id);

/*
 * SetPrivateIntPending - mark a private (SGI/PPI) interrupt as pending
 *
 * Inputs
 *
 *  gicr - which Redistributor to program
 *
 *  id - interrupt identifier
 *
 * Returns
 *
 *   <nothing>
 */
void SetPrivateIntPending(uint32_t gicr, uint32_t id);

/*
 * ClearPrivateIntPending - mark a private (SGI/PPI) interrupt as not pending
 *
 * Inputs
 *
 *  gicr - which Redistributor to program
 *
 *  id - interrupt identifier
 *
 * Returns
 *
 *   <nothing>
 */
void ClearPrivateIntPending(uint32_t gicr, uint32_t id);

/*
 * GetPrivateIntPending - query whether a private (SGI/PPI) interrupt is pending
 *
 * Inputs
 *
 *  gicr - which Redistributor to program
 *
 *  id - interrupt identifier
 *
 * Returns
 *
 *   pending status
 */
uint32_t GetPrivateIntPending(uint32_t gicr, uint32_t id);

/*
 * SetPrivateIntSecurity - mark a private (SGI/PPI) interrupt as
 *     security <group>
 *
 * Inputs
 *
 *  gicr - which Redistributor to program
 *
 *  id - which interrupt to mark
 *
 *  group - the group for the interrupt
 *
 * Returns
 *
 *  <nothing>
 */
```

```
void SetPrivateIntSecurity(uint32_t gicr, uint32_t id, GICIGROUPRBits_t group);

/*
 * SetPrivateIntSecurityBlock - mark all 32 private (SGI/PPI)
 *     interrupts as security <group>
 *
 * Inputs:
 *
 *  gicr - which Redistributor to program
 *
 *  group - the group for the interrupt
 *
 * Returns:
 *
 *  <nothing>
 */
void SetPrivateIntSecurityBlock(uint32_t gicr, GICIGROUPRBits_t group);

#endif /* ndef GICV3_h */

/* EOF GICv3.h */
```

Create the file `GICv3_gicc.h` containing the following code:

```
/*
 * GICv3_gicc.h - prototypes and inline functions for GICC system register
 operations
 *
 * Copyright (c) 2014-2017 Arm Limited (or its affiliates). All rights reserved.
 * Use, modification and redistribution of this file is subject to your possession
 of a
 * valid End User License Agreement for the Arm Product of which these examples are
 part of
 * and your compliance with all applicable terms and conditions of such licence
 agreement.
 */
#ifndef GICV3_gicc_h
#define GICV3_gicc_h

/*******************************************************************************/

typedef enum
{
    sreSRE = (1 << 0),
    sreDFB = (1 << 1),
    sreDIB = (1 << 2),
    sreEnable = (1 << 3)
} ICC_SREBits_t;

static inline void setICC_SRE_EL1(ICC_SREBits_t mode)
{
    asm("msr  ICC_SRE_EL1, %0\n; isb" :: "r" ((uint64_t)mode));
}

static inline uint64_t getICC_SRE_EL1(void)
{
    uint64_t retc;

    asm("mrs  %0, ICC_SRE_EL1\n" : "=r" (retc));

    return retc;
}

static inline void setICC_SRE_EL2(ICC_SREBits_t mode)
{
    asm("msr  ICC_SRE_EL2, %0\n; isb" :: "r" ((uint64_t)mode));
}
```

```
static inline uint64_t getICC_SRE_EL2(void)
{
    uint64_t retc;

    asm("mrs  %0, ICC_SRE_EL2\n" : "=r" (retc));

    return retc;
}

static inline void setICC_SRE_EL3(ICC_SREBits_t mode)
{
    asm("msr  ICC_SRE_EL3, %0\n; isb" :: "r" ((uint64_t)mode));
}

static inline uint64_t getICC_SRE_EL3(void)
{
    uint64_t retc;

    asm("mrs  %0, ICC_SRE_EL3\n" : "=r" (retc));

    return retc;
}

/********************************************************************/

typedef enum
{
    igrpEnable = (1 << 0),
    igrpEnableGrp1NS = (1 << 0),
    igrpEnableGrp1S = (1 << 2)
} ICC_IGRPBits_t;

static inline void setICC_IGRPEN0_EL1(ICC_IGRPBits_t mode)
{
    asm("msr  ICC_IGRPEN0_EL1, %0\n; isb" :: "r" ((uint64_t)mode));
}

static inline void setICC_IGRPEN1_EL1(ICC_IGRPBits_t mode)
{
    asm("msr  ICC_IGRPEN1_EL1, %0\n; isb" :: "r" ((uint64_t)mode));
}

static inline void setICC_IGRPEN1_EL3(ICC_IGRPBits_t mode)
{
    asm("msr  ICC_IGRPEN1_EL3, %0\n; isb" :: "r" ((uint64_t)mode));
}

/********************************************************************/

typedef enum
{
    ctlrCBPR = (1 << 0),
    ctlrCBPR_EL1S = (1 << 0),
    ctlrEOImode = (1 << 1),
    ctlrCBPR_EL1NS = (1 << 1),
    ctlrEOImode_EL3 = (1 << 2),
    ctlrEOImode_EL1S = (1 << 3),
    ctlrEOImode_EL1NS = (1 << 4),
    ctlrRM = (1 << 5),
    ctlrPMHE = (1 << 6)
} ICC_CTLRBits_t;

static inline void setICC_CTLR_EL1(ICC_CTLRBits_t mode)
{
    asm("msr  ICC_CTLR_EL1, %0\n; isb" :: "r" ((uint64_t)mode));
}

static inline uint64_t getICC_CTLR_EL1(void)
{
    uint64_t retc;
```

```
    asm("mrs  %0, ICC_CTLR_EL1\n" : "=r" (retc));

    return retc;
}

static inline void setICC_CTLR_EL3(ICC_CTLRBits_t mode)
{
    asm("msr  ICC_CTLR_EL3, %0\n; isb" :: "r" ((uint64_t)mode));
}

static inline uint64_t getICC_CTLR_EL3(void)
{
    uint64_t retc;

    asm("mrs  %0, ICC_CTLR_EL3\n" : "=r" (retc));

    return retc;
}

/*******************************************************************/

static inline uint64_t getICC_IAR0(void)
{
    uint64_t retc;

    asm("mrs  %0, ICC_IAR0_EL1\n" : "=r" (retc));

    return retc;
}

static inline uint64_t getICC_IAR1(void)
{
    uint64_t retc;

    asm("mrs  %0, ICC_IAR1_EL1\n" : "=r" (retc));

    return retc;
}

static inline void setICC_EOIR0(uint32_t interrupt)
{
    asm("msr  ICC_EOIR0_EL1, %0\n; isb" :: "r" ((uint64_t)interrupt));
}

static inline void setICC_EOIR1(uint32_t interrupt)
{
    asm("msr  ICC_EOIR1_EL1, %0\n; isb" :: "r" ((uint64_t)interrupt));
}

static inline void setICC_DIR(uint32_t interrupt)
{
    asm("msr  ICC_DIR_EL1, %0\n; isb" :: "r" ((uint64_t)interrupt));
}

static inline void setICC_PMR(uint32_t priority)
{
    asm("msr  ICC_PMR_EL1, %0\n; isb" :: "r" ((uint64_t)priority));
}

static inline void setICC_BPR0(uint32_t binarypoint)
{
    asm("msr  ICC_BPR0_EL1, %0\n; isb" :: "r" ((uint64_t)binarypoint));
}

static inline void setICC_BPR1(uint32_t binarypoint)
{
    asm("msr  ICC_BPR1_EL1, %0\n; isb" :: "r" ((uint64_t)binarypoint));
}

static inline uint64_t getICC_BPR0(void)
{
```

```
    uint64_t retc;

    asm("mrs  %0, ICC_BPR0_EL1\n" : "=r" (retc));

    return retc;
}

static inline uint64_t getICC_BPR1(void)
{
    uint64_t retc;

    asm("mrs  %0, ICC_BPR1_EL1\n" : "=r" (retc));

    return retc;
}

static inline uint64_t getICC_RPR(void)
{
    uint64_t retc;

    asm("mrs  %0, ICC_RPR_EL1\n" : "=r" (retc));

    return retc;
}

/*****************************************************************/

typedef enum
{
    sgirIRMTarget = 0,
    sgirIRMAll = (1ull << 40)
} ICC_SGIRBits_t;

static inline void setICC_SGI0R(uint8_t aff3, uint8_t aff2,
                uint8_t aff1, ICC_SGIRBits_t irm,
                uint16_t targetlist, uint8_t intid)
{
    uint64_t packedbits = (((uint64_t)aff3 << 48) | ((uint64_t)aff2 << 32) | \
                ((uint64_t)aff1 << 16) | irm | targetlist | \
                ((uint64_t)(intid & 0x0f) << 24));

    asm("msr  ICC_SGI0R_EL1, %0\n; isb" :: "r" (packedbits));
}

static inline void setICC_SGI1R(uint8_t aff3, uint8_t aff2,
                uint8_t aff1, ICC_SGIRBits_t irm,
                uint16_t targetlist, uint8_t intid)
{
    uint64_t packedbits = (((uint64_t)aff3 << 48) | ((uint64_t)aff2 << 32) | \
                ((uint64_t)aff1 << 16) | irm | targetlist | \
                ((uint64_t)(intid & 0x0f) << 24));

    asm("msr  ICC_SGI1R_EL1, %0\n; isb" :: "r" (packedbits));
}

static inline void setICC_ASGI1R(uint8_t aff3, uint8_t aff2,
                uint8_t aff1, ICC_SGIRBits_t irm,
                uint16_t targetlist, uint8_t intid)
{
    uint64_t packedbits = (((uint64_t)aff3 << 48) | ((uint64_t)aff2 << 32) | \
                ((uint64_t)aff1 << 16) | irm | targetlist | \
                ((uint64_t)(intid & 0x0f) << 24));

    asm("msr  ICC_ASGI1R_EL1, %0\n; isb" :: "r" (packedbits));
}

#endif /* ndef GICV3_gicc_h */
```

Create the file `GICv3_gicd.c` containing the following code:

```
/*
 * GICv3_gicd.c - generic driver code for GICv3 distributor
 *
 * Copyright (c) 2014-2017 Arm Limited (or its affiliates). All rights reserved.
 * Use, modification and redistribution of this file is subject to your possession
 of a
 * valid End User License Agreement for the Arm Product of which these examples are
 part of
 * and your compliance with all applicable terms and conditions of such licence
 agreement.
 */
#include <stdint.h>

#include "GICv3.h"

typedef struct
{
        volatile uint32_t GICD_CTLR;              // +0x0000
  const volatile uint32_t GICD_TYPER;             // +0x0004
  const volatile uint32_t GICD_IIDR;              // +0x0008

  const volatile uint32_t padding0;               // +0x000c

        volatile uint32_t GICD_STATUSR;           // +0x0010

  const volatile uint32_t padding1[3];            // +0x0014

        volatile uint32_t IMP_DEF[8];             // +0x0020

        volatile uint32_t GICD_SETSPI_NSR;        // +0x0040
  const volatile uint32_t padding2;               // +0x0044
        volatile uint32_t GICD_CLRSPI_NSR;        // +0x0048
  const volatile uint32_t padding3;               // +0x004c
        volatile uint32_t GICD_SETSPI_SR;         // +0x0050
  const volatile uint32_t padding4;               // +0x0054
        volatile uint32_t GICD_CLRSPI_SR;         // +0x0058

  const volatile uint32_t padding5[3];            // +0x005c

        volatile uint32_t GICD_SEIR;              // +0x0068

  const volatile uint32_t padding6[5];            // +0x006c

        volatile uint32_t GICD_IGROUPR[32];       // +0x0080

        volatile uint32_t GICD_ISENABLER[32];     // +0x0100
        volatile uint32_t GICD_ICENABLER[32];     // +0x0180
        volatile uint32_t GICD_ISPENDR[32];       // +0x0200
        volatile uint32_t GICD_ICPENDR[32];       // +0x0280
        volatile uint32_t GICD_ISACTIVER[32];     // +0x0300
        volatile uint32_t GICD_ICACTIVER[32];     // +0x0380

        volatile uint8_t  GICD_IPRIORITYR[1024];  // +0x0400
        volatile uint8_t  GICD_ITARGETSR[1024];   // +0x0800
        volatile uint32_t GICD_ICFGR[64];         // +0x0c00
        volatile uint32_t GICD_IGRPMODR[32];      // +0x0d00
  const volatile uint32_t padding7[32];           // +0x0d80
        volatile uint32_t GICD_NSACR[64];         // +0x0e00

        volatile uint32_t GICD_SGIR;              // +0x0f00

  const volatile uint32_t padding8[3];            // +0x0f04

        volatile uint32_t GICD_CPENDSGIR[4];      // +0x0f10
        volatile uint32_t GICD_SPENDSGIR[4];      // +0x0f20

  const volatile uint32_t padding9[52];           // +0x0f30
```

```
  const volatile uint32_t padding10[5120];        // +0x1000

        volatile uint64_t GICD_IROUTER[1024];     // +0x6000
} GICv3_distributor;

/*
 * use the scatter file to place GICD
 */
static GICv3_distributor __attribute__((section(".bss.distributor"))) gicd;

void ConfigGICD(GICDCTLRFlags_t flags)
{
    gicd.GICD_CTLR = flags;
}

void EnableGICD(GICDCTLRFlags_t flags)
{
    gicd.GICD_CTLR |= flags;
}

void DisableGICD(GICDCTLRFlags_t flags)
{
    gicd.GICD_CTLR &= ~flags;
}

void SyncAREinGICD(GICDCTLRFlags_t flags, uint32_t dosync)
{
    if (dosync)
    {
 const uint32_t tmask = gicdctlr_ARE_S | gicdctlr_ARE_NS;
 const uint32_t tval = flags & tmask;

 while ((gicd.GICD_CTLR & tmask) != tval)
     continue;
    }
    else
 gicd.GICD_CTLR = flags;
}

void EnableSPI(uint32_t id)
{
    uint32_t bank;

    /*
     * GICD_ISENABLER has 32 interrupts for each register
     */
    bank = (id >> 5) & RANGE_LIMIT(gicd.GICD_ISENABLER);
    id &= 32 - 1;

    gicd.GICD_ISENABLER[bank] = 1 << id;

    return;
}

void DisableSPI(uint32_t id)
{
    uint32_t bank;

    /*
     * GICD_ISENABLER has 32 interrupts for each register
     */
    bank = (id >> 5) & RANGE_LIMIT(gicd.GICD_ICENABLER);
    id &= 32 - 1;

    gicd.GICD_ICENABLER[bank] = 1 << id;

    return;
}

void SetSPIPriority(uint32_t id, uint32_t priority)
{
```

```
    uint32_t bank;

    /*
     * GICD_IPRIORITYR has one byte-wide entry for each interrupt
     */
    bank = id & RANGE_LIMIT(gicd.GICD_IPRIORITYR);

    gicd.GICD_IPRIORITYR[bank] = priority;
}

uint32_t GetSPIPriority(uint32_t id)
{
    uint32_t bank;

    /*
     * GICD_IPRIORITYR has one byte-wide entry for each interrupt
     */
    bank = id & RANGE_LIMIT(gicd.GICD_IPRIORITYR);

    return (uint32_t)(gicd.GICD_IPRIORITYR[bank]);
}

void SetSPIRoute(uint32_t id, uint64_t affinity, GICDIROUTERBits_t mode)
{
    uint32_t bank;

    /*
     * GICD_IROUTER has one doubleword-wide entry for each interrupt
     */
    bank = id & RANGE_LIMIT(gicd.GICD_IROUTER);

    gicd.GICD_IROUTER[bank] = affinity | (uint64_t)mode;
}

uint64_t GetSPIRoute(uint32_t id)
{
    uint32_t bank;

    /*
     * GICD_IROUTER has one doubleword-wide entry for each interrupt
     */
    bank = id & RANGE_LIMIT(gicd.GICD_IROUTER);

    return gicd.GICD_IROUTER[bank];
}

void SetSPITarget(uint32_t id, uint32_t target)
{
    uint32_t bank;

    /*
     * GICD_ITARGETSR has one byte-wide entry for each interrupt
     */
    bank = id & RANGE_LIMIT(gicd.GICD_ITARGETSR);

    gicd.GICD_ITARGETSR[bank] = target;
}

uint32_t GetSPITarget(uint32_t id)
{
    uint32_t bank;

    /*
     * GICD_ITARGETSR has one byte-wide entry for each interrupt
     */
    /*
     * GICD_ITARGETSR has 4 interrupts for each register. That is, 8-bits of
     * target bitmap for each register
     */
    bank = id & RANGE_LIMIT(gicd.GICD_ITARGETSR);
```

```
    return (uint32_t)(gicd.GICD_ITARGETSR[bank]);
}

void ConfigureSPI(uint32_t id, GICDICFGRBits_t config)
{
    uint32_t bank, tmp;

    /*
     * GICD_ICFGR has 16 interrupts for each register. That is, 2-bits of
     * configuration for each register
     */
    bank = (id >> 4) & RANGE_LIMIT(gicd.GICD_ICFGR);
    config &= 3;

    id = (id & 0xf) << 1;

    tmp = gicd.GICD_ICFGR[bank];
    tmp &= ~(3 << id);
    tmp |= config << id;
    gicd.GICD_ICFGR[bank] = tmp;
}

void SetSPIPending(uint32_t id)
{
    uint32_t bank;

    /*
     * GICD_ISPENDR has 32 interrupts for each register
     */
    bank = (id >> 5) & RANGE_LIMIT(gicd.GICD_ISPENDR);
    id &= 0x1f;

    gicd.GICD_ISPENDR[bank] = 1 << id;
}

void ClearSPIPending(uint32_t id)
{
    uint32_t bank;

    /*
     * GICD_ICPENDR has 32 interrupts for each register
     */
    bank = (id >> 5) & RANGE_LIMIT(gicd.GICD_ICPENDR);
    id &= 0x1f;

    gicd.GICD_ICPENDR[bank] = 1 << id;
}

uint32_t GetSPIPending(uint32_t id)
{
    uint32_t bank;

    /*
     * GICD_ICPENDR has 32 interrupts for each register
     */
    bank = (id >> 5) & RANGE_LIMIT(gicd.GICD_ICPENDR);
    id &= 0x1f;

    return (gicd.GICD_ICPENDR[bank] >> id) & 1;
}

void SetSPISecurity(uint32_t id, GICIGROUPRBits_t group)
{
    uint32_t bank, groupmod;

    /*
     * GICD_IGROUPR has 32 interrupts for each register
     */
    bank = (id >> 5) & RANGE_LIMIT(gicd.GICD_IGROUPR);
    id &= 0x1f;
```

```
    /*
     * the single group argument is split into two separate
     * registers, so filter out and remove the (new to gicv3)
     * group modifier bit
     */
    groupmod = (group >> 1) & 1;
    group &= 1;

    /*
     * either set or clear the Group bit for the interrupt as appropriate
     */
    if (group)
 gicd.GICD_IGROUPR[bank] |= 1 << id;
    else
 gicd.GICD_IGROUPR[bank] &= ~(1 << id);

    /*
     * now deal with groupmod
     */
    if (groupmod)
 gicd.GICD_IGRPMODR[bank] |= 1 << id;
    else
 gicd.GICD_IGRPMODR[bank] &= ~(1 << id);
}

void SetSPISecurityBlock(uint32_t block, GICIGROUPRBits_t group)
{
    uint32_t groupmod;
    const uint32_t nbits = (sizeof group * 8) - 1;

    /*
     * GICD_IGROUPR has 32 interrupts for each register
     */
    block &= RANGE_LIMIT(gicd.GICD_IGROUPR);

    /*
     * get each bit of group config duplicated over all 32-bits in a word
     */
    groupmod = (uint32_t)(((int32_t)group << (nbits - 1)) >> 31);
    group = (uint32_t)(((int32_t)group << nbits) >> 31);

    /*
     * set the security state for this block of SPIs
     */
    gicd.GICD_IGROUPR[block] = group;
    gicd.GICD_IGRPMODR[block] = groupmod;
}

void SetSPISecurityAll(GICIGROUPRBits_t group)
{
    uint32_t block;

    /*
     * GICD_TYPER.ITLinesNumber gives (No. SPIS / 32) - 1, and we
     * want to iterate over all blocks excluding 0 (which are the
     * SGI/PPI interrupts, and not relevant here)
     */
    for (block = (gicd.GICD_TYPER & ((1 << 5) - 1)); block > 0; --block)
 SetSPISecurityBlock(block, group);
}

/* EOF GICv3_gicd.c */
```

Create the file `GICv3_gicr.c` containing the following code:

```
/*
 * GICv3_gicr.c - generic driver code for GICv3 redistributor
 *
```

```
 * Copyright (c) 2014-2020 Arm Limited (or its affiliates). All rights reserved.
 * Use, modification and redistribution of this file is subject to your possession
 of a
 * valid End User License Agreement for the Arm Product of which these examples are
 part of
 * and your compliance with all applicable terms and conditions of such licence
 agreement.
 */
#include "GICv3.h"

/*
 * Physical LPI Redistributor register map
 */
typedef struct
{
        volatile uint32_t GICR_CTLR;             // +0x0000 - RW - Redistributor
 Control Register
  const volatile uint32_t GICR_IIDR;             // +0x0004 - RO - Implementer
 Identification Register
  const volatile uint32_t GICR_TYPER[2];         // +0x0008 - RO - Redistributor
 Type Register
        volatile uint32_t GICR_STATUSR;          // +0x0010 - RW - Error Reporting
 Status Register, optional
        volatile uint32_t GICR_WAKER;            // +0x0014 - RW - Redistributor
 Wake Register
  const volatile uint32_t padding1[2];           // +0x0018 - RESERVED
#ifndef USE_GIC600
        volatile uint32_t IMPDEF1[8];            // +0x0020 - ?? - IMPLEMENTATION
 DEFINED
#else
        volatile uint32_t GICR_FCTLR;            // +0x0020 - RW - Function Control
 Register
        volatile uint32_t GICR_PWRR;             // +0x0024 - RW - Power Management
 Control Register
        volatile uint32_t GICR_CLASS;            // +0x0028 - RW - Class Register
  const volatile uint32_t padding2[5];           // +0x002C - RESERVED
#endif
        volatile uint64_t GICR_SETLPIR;          // +0x0040 - WO - Set LPI Pending
 Register
        volatile uint64_t GICR_CLRLPIR;          // +0x0048 - WO - Clear LPI Pending
 Register
  const volatile uint32_t padding3[8];           // +0x0050 - RESERVED
        volatile uint64_t GICR_PROPBASER;        // +0x0070 - RW - Redistributor
 Properties Base Address Register
        volatile uint64_t GICR_PENDBASER;        // +0x0078 - RW - Redistributor LPI
 Pending Table Base Address Register
  const volatile uint32_t padding4[8];           // +0x0080 - RESERVED
        volatile uint64_t GICR_INVLPIR;          // +0x00A0 - WO - Redistributor
 Invalidate LPI Register
  const volatile uint32_t padding5[2];           // +0x00A8 - RESERVED
        volatile uint64_t GICR_INVALLR;          // +0x00B0 - WO - Redistributor
 Invalidate All Register
  const volatile uint32_t padding6[2];           // +0x00B8 - RESERVED
        volatile uint64_t GICR_SYNCR;            // +0x00C0 - RO - Redistributor
 Synchronize Register
  const volatile uint32_t padding7[2];           // +0x00C8 - RESERVED
  const volatile uint32_t padding8[12];          // +0x00D0 - RESERVED
        volatile uint64_t IMPDEF2;               // +0x0100 - WO - IMPLEMENTATION
 DEFINED
  const volatile uint32_t padding9[2];           // +0x0108 - RESERVED
        volatile uint64_t IMPDEF3;               // +0x0110 - WO - IMPLEMENTATION
 DEFINED
  const volatile uint32_t padding10[2];          // +0x0118 - RESERVED
} GICv3_redistributor_RD;

/*
 * SGI and PPI Redistributor register map
 */
typedef struct
{
  const volatile uint32_t padding1[32];          // +0x0000 - RESERVED
```

```
        volatile uint32_t GICR_IGROUPR0;         // +0x0080 - RW - Interrupt Group
 Registers (Security Registers in GICv1)
  const volatile uint32_t padding2[31];          // +0x0084 - RESERVED
        volatile uint32_t GICR_ISENABLER;        // +0x0100 - RW - Interrupt Set-
Enable Registers
  const volatile uint32_t padding3[31];          // +0x0104 - RESERVED
        volatile uint32_t GICR_ICENABLER;        // +0x0180 - RW - Interrupt Clear-
Enable Registers
  const volatile uint32_t padding4[31];          // +0x0184 - RESERVED
        volatile uint32_t GICR_ISPENDR;          // +0x0200 - RW - Interrupt Set-
Pending Registers
  const volatile uint32_t padding5[31];          // +0x0204 - RESERVED
        volatile uint32_t GICR_ICPENDR;          // +0x0280 - RW - Interrupt Clear-
Pending Registers
  const volatile uint32_t padding6[31];          // +0x0284 - RESERVED
        volatile uint32_t GICR_ISACTIVER;        // +0x0300 - RW - Interrupt Set-
Active Register
  const volatile uint32_t padding7[31];          // +0x0304 - RESERVED
        volatile uint32_t GICR_ICACTIVER;        // +0x0380 - RW - Interrupt Clear-
Active Register
  const volatile uint32_t padding8[31];          // +0x0184 - RESERVED
        volatile uint8_t  GICR_IPRIORITYR[32];   // +0x0400 - RW - Interrupt
 Priority Registers
  const volatile uint32_t padding9[504];         // +0x0420 - RESERVED
        volatile uint32_t GICR_ICnoFGR[2];       // +0x0C00 - RW - Interrupt
 Configuration Registers
  const volatile uint32_t padding10[62];         // +0x0C08 - RESERVED
        volatile uint32_t GICR_IGRPMODR0;        // +0x0D00 - RW - ????
  const volatile uint32_t padding11[63];         // +0x0D04 - RESERVED
        volatile uint32_t GICR_NSACR;            // +0x0E00 - RW - Non-Secure Access
 Control Register
} GICv3_redistributor_SGI;

/*
 * We have a multiplicity of GIC Redistributors; on the GIC-AEM and
 * GIC-500 they are arranged as one 128KB region per redistributor: one
 * 64KB page of GICR LPI registers, and one 64KB page of GICR Private
 * Int registers
 */
typedef struct
{
    union
    {
        GICv3_redistributor_RD RD_base;
        uint8_t padding[64 * 1024];
    } RDblock;

    union
    {
        GICv3_redistributor_SGI SGI_base;
        uint8_t padding[64 * 1024];
    } SGIblock;
} GICv3_GICR;

/*
 * use the scatter file to place GIC Redistributor base address
 *
 * although this code does not know how many Redistributor banks
 * a particular system will have, we declare gicrbase as an array
 * to avoid unwanted compiler optimizations when calculating the
 * base of a particular Redistributor bank
 */
static const GICv3_GICR gicrbase[2] __attribute__((section (".bss.redistributor")));

/*******************************************************************/

/*
 * utility functions to calculate base of a particular
 * Redistributor bank
 */
```

```
static inline GICv3_redistributor_RD *const getgicrRD(uint32_t gicr)
{
    GICv3_GICR *const arraybase = (GICv3_GICR *const)&gicrbase;

    return &((arraybase + gicr)->RDblock.RD_base);
}

static inline GICv3_redistributor_SGI *const getgicrSGI(uint32_t gicr)
{
    GICv3_GICR *arraybase = (GICv3_GICR *)(&gicrbase);

    return &(arraybase[gicr].SGIblock.SGI_base);
}

/*********************************************************************/

// This function walks a block of RDs to find one with the matching affinity
uint32_t GetGICR(uint32_t affinity)
{
  GICv3_redistributor_RD* gicr;
  uint32_t index = 0;

  do
  {
    gicr = getgicrRD(index);
    if (gicr->GICR_TYPER[1] == affinity)
       return index;

    index++;
  }
  while((gicr->GICR_TYPER[0] & (1<<4)) == 0); // Keep looking until GICR_TYPER.Last
 reports no more RDs in block

  return 0xFFFFFFFF; // return -1 to signal not RD found
}

void WakeupGICR(uint32_t gicr)
{
    GICv3_redistributor_RD *const gicrRD = getgicrRD(gicr);

#ifdef USE_GIC600
/* GICR_PWRR fields */
#define PWRR_RDPD_SHIFT  0
#define PWRR_RDAG_SHIFT  1
#define PWRR_RDGPD_SHIFT 2
#define PWRR_RDGPO_SHIFT 3

#define PWRR_RDPD (1 << PWRR_RDPD_SHIFT)
#define PWRR_RDAG (1 << PWRR_RDAG_SHIFT)
#define PWRR_RDGPD (1 << PWRR_RDGPD_SHIFT)
#define PWRR_RDGPO (1 << PWRR_RDGPO_SHIFT)

/*
 * Values to write to GICR_PWRR register to power redistributor
 * for operating through the core (GICR_PWRR.RDAG = 0)
 */
#define PWRR_ON  (0 << PWRR_RDPD_SHIFT)
#define PWRR_OFF (1 << PWRR_RDPD_SHIFT)

    do {
     while (((gicrRD->GICR_PWRR & PWRR_RDGPD) >> PWRR_RDGPD_SHIFT) != ((gicrRD-
>GICR_PWRR & PWRR_RDGPO) >> PWRR_RDGPO_SHIFT));
     /* Power on redistributor */
     gicrRD->GICR_PWRR=PWRR_ON;

    } while ((gicrRD->GICR_PWRR & PWRR_RDPD) != PWRR_ON);
#endif

    /*
     * step 1 - ensure GICR_WAKER.ProcessorSleep is off
     */
```

```
    gicrRD->GICR_WAKER &= ~gicrwaker_ProcessorSleep;

    /*
     * step 2 - wait for children asleep to be cleared
     */
    while ((gicrRD->GICR_WAKER & gicrwaker_ChildrenAsleep) != 0)
        continue;

    /*
     * OK, GICR is go
     */
    return;
}

void EnablePrivateInt(uint32_t gicr, uint32_t id)
{
    GICv3_redistributor_SGI *const gicrSGI = getgicrSGI(gicr);

    id &= 0x1f;

    gicrSGI->GICR_ISENABLER = 1 << id;
}

void DisablePrivateInt(uint32_t gicr, uint32_t id)
{
    GICv3_redistributor_SGI *const gicrSGI = getgicrSGI(gicr);

    id &= 0x1f;

    gicrSGI->GICR_ICENABLER = 1 << id;
}

void SetPrivateIntPriority(uint32_t gicr, uint32_t id, uint32_t priority)
{
    GICv3_redistributor_SGI *const gicrSGI = getgicrSGI(gicr);

    /*
     * GICD_IPRIORITYR has one byte-wide entry per interrupt
     */
    id &= RANGE_LIMIT(gicrSGI->GICR_IPRIORITYR);

    gicrSGI->GICR_IPRIORITYR[id] = priority;
}

uint32_t GetPrivateIntPriority(uint32_t gicr, uint32_t id)
{
    GICv3_redistributor_SGI *const gicrSGI = getgicrSGI(gicr);

    /*
     * GICD_IPRIORITYR has one byte-wide entry per interrupt
     */
    id &= RANGE_LIMIT(gicrSGI->GICR_IPRIORITYR);

    return (uint32_t)(gicrSGI->GICR_IPRIORITYR[id]);
}

void SetPrivateIntPending(uint32_t gicr, uint32_t id)
{
    GICv3_redistributor_SGI *const gicrSGI = getgicrSGI(gicr);

    /*
     * GICR_ISPENDR is one 32-bit register
     */
    id &= 0x1f;

    gicrSGI->GICR_ISPENDR = 1 << id;
}

void ClearPrivateIntPending(uint32_t gicr, uint32_t id)
{
    GICv3_redistributor_SGI *const gicrSGI = getgicrSGI(gicr);
```

```
    /*
     * GICR_ICPENDR is one 32-bit register
     */
    id &= 0x1f;

    gicrSGI->GICR_ICPENDR = 1 << id;
}

uint32_t GetPrivateIntPending(uint32_t gicr, uint32_t id)
{
    GICv3_redistributor_SGI *const gicrSGI = getgicrSGI(gicr);

    /*
     * GICR_ISPENDR is one 32-bit register
     */
    id &= 0x1f;

    return (gicrSGI->GICR_ISPENDR >> id) & 0x01;
}

void SetPrivateIntSecurity(uint32_t gicr, uint32_t id, GICIGROUPRBits_t group)
{
    GICv3_redistributor_SGI *const gicrSGI = getgicrSGI(gicr);
    uint32_t groupmod;

    /*
     * GICR_IGROUPR0 is one 32-bit register
     */
    id &= 0x1f;

    /*
     * the single group argument is split into two separate
     * registers, so filter out and remove the (new to gicv3)
     * group modifier bit
     */
    groupmod = (group >> 1) & 1;
    group &= 1;

    /*
     * either set or clear the Group bit for the interrupt as appropriate
     */
    if (group)
        gicrSGI->GICR_IGROUPR0 |= 1 << id;
    else
        gicrSGI->GICR_IGROUPR0 &= ~(1 << id);

    /*
     * now deal with groupmod
     */
    if (groupmod)
        gicrSGI->GICR_IGRPMODR0 |= 1 << id;
    else
        gicrSGI->GICR_IGRPMODR0 &= ~(1 << id);
}

void SetPrivateIntSecurityBlock(uint32_t gicr, GICIGROUPRBits_t group)
{
    GICv3_redistributor_SGI *const gicrSGI = getgicrSGI(gicr);
    const uint32_t nbits = (sizeof group * 8) - 1;
    uint32_t groupmod;

    /*
     * get each bit of group config duplicated over all 32 bits
     */
    groupmod = (uint32_t)(((int32_t)group << (nbits - 1)) >> 31);
    group = (uint32_t)(((int32_t)group << nbits) >> 31);

    /*
     * set the security state for this block of SPIs
     */
```

```
    gicrSGI->GICR_IGROUPR0 = group;
    gicrSGI->GICR_IGRPMODR0 = groupmod;
}

/* EOF GICv3_gicr.c */
```

Create the file `main.c` containing the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>

// We use attributes to force the compiler to use the local-exec model
// Alternatively, you can compile with -ftls-model=local-exec
// Global TLS RW variable.
__thread int foo __attribute__((tls_model("local-exec"))) = 0xdeadbeef;
// Global TLS ZI variable
__thread int bar __attribute__((tls_model("local-exec"))) = 0;


// You must implement this function. The register used here must
// match the one specified with -mtp=<el> during compilation.
// Defining this function as always inline and static with inline
// assembly means it only ever uses one instruction without needing
// a full function call.
__attribute__((always_inline)) static void write_tp(void* tls_data)
{
    __asm volatile("msr TPIDR_EL0, %0" : : "r"(tls_data) : "cc");
}

// Example function to initialize TLS data from memory.
// This function assumes that a single-threaded application is being used.
void __attribute((noinline)) initialise_tls_from_mem(void *data_start,
                                                     size_t data_length,
                                                     size_t bss_length)
{
    // Reserve space for the thread's TLS. The ABI requires it to be in
    // form.
    // | 8-bytes | 8-bytes  | sizeof applications TLS |
    // | TCB     | reserved | TLS data                |
    // The TCB points to the Thread Control Block. As we are a single threaded
    // application using local exec only we do not need one as all TLS can be
    // accessed via offsets from the thread pointer which points to the start
    // of the structure above.
    void *app_tls = malloc(8 /* TCB */ +
                           8 /* reserved */ +
                           data_length /* .tdata */ +
                           bss_length /* .tbss */);
    if (app_tls == NULL) {
        printf("Malloc of TLS data failed\n");
        exit(1);
    }
    // copy .tdata from template to TLS data
    memcpy(app_tls + 16 /* data starts after TCB and reserved */,
           data_start,
           data_length);
    // .tbss starts after .tdata, initialise with 0
    memset(app_tls + 16 + data_length, 0, bss_length);
    // Set thread pointer to point to our copy of the TLS data.
    write_tp(app_tls);
    // We can now use local exec TLS variables
}

// Linker-defined symbols for accessing information about
// the local and size of TLS RW and ZI data.
extern int Image$$ER_TLS_RW$$Base;
extern int Image$$ER_TLS_RW$$Limit;
```

```
extern int Image$$ER_TLS_ZI$$ZI$$Limit;

// Simple program that creates the TLS from the TLS template then prints
// out the values using TLS variables.
// This only works for local-exec TLS as all TLS variables are at a known
// offset from the thread pointer register.
int main(void) {
    initialise_tls_from_mem(
        // Start address of TLS RW data
        (unsigned int*)&Image$$ER_TLS_RW$$Base,
        // Number of bytes of TLS RW data
        (size_t)&Image$$ER_TLS_RW$$Limit - (size_t)&Image$$ER_TLS_RW$$Base,
        // Number of bytes of TLS ZI data
        (size_t)&Image$$ER_TLS_ZI$$ZI$$Limit - (size_t)&Image$$ER_TLS_RW$$Limit
    );

    printf("TLS RW foo @ 0x%p = 0x%x\n", &foo, foo);
    printf("TLS ZI bar @ 0x%p = 0x%x\n", &bar, bar);

    return 0;
}
```

Create the file `retarget.c` containing the following code:

```
/*
** Copyright (c) 2006-2014 Arm Limited (or its affiliates). All rights reserved.
** Use, modification and redistribution of this file is subject to your possession
 of a
** valid End User License Agreement for the Arm Product of which these examples are
 part of
** and your compliance with all applicable terms and conditions of such licence
 agreement.
*/

/*
** This file contains re-implementations of functions whose
** C library implementations rely on semihosting.
**
** Define USE_SERIAL_PORT to retarget the I/O only to the serial port.
** Otherwise, I/O is targeted to the debugger console using semihosting.
**
** Define STANDALONE to eliminate all use of semihosting-using functions too.
*/

#include <stdio.h>


#define TRUE 1
#define FALSE 0



/*
** Importing __use_no_semihosting ensures that our image doesn't link
** with any C Library code that makes direct use of semihosting.
**
** Build with STANDALONE to include this symbol.
*/

#ifdef STANDALONE
#define USE_SERIAL_PORT 1
asm(".global __use_no_semihosting");
#endif


/*
** Retargeted I/O
** ==============
```

```
** The following C library functions make use of semihosting
** to read or write characters to the debugger console: fputc(),
** fgetc(), and _ttywrch().  They must be retargeted to write to
** the model's UART.  __backspace() must also be retargeted
** with this layer to enable scanf().  See the Compiler and
** Libraries Guide.
*/

#ifdef USE_SERIAL_PORT

extern void uart_putc_polled(char c);
extern char uart_getchar_polled(void);

/*
** These must be defined to avoid linking in stdio.o from the
** C Library
*/

struct __FILE { int handle;   /* Add whatever you need here */};
FILE __stdout;
FILE __stdin;

/*
** __backspace must return the last char read to the stream
** fgetc() needs to keep a record of whether __backspace was
** called directly before it
*/
int last_char_read;
int backspace_called;


int fgetc(FILE *f)
{
    unsigned char tempch;
    tempch = uart_getchar_polled();
    last_char_read = (int)tempch;       /* backspace must return this value */
    return tempch;
}


int fputc(int ch, FILE *f)
{
    unsigned char tempch = ch;
    if (tempch == '\n') uart_putc_polled('\r');
    uart_putc_polled(tempch);
    return ch;
}


void _ttywrch(int ch)
{
    unsigned char tempch = ch;
    if (tempch == '\n') uart_putc_polled('\r');
    uart_putc_polled(tempch);
}

/*
** The effect of __backspace() should be to return the last character
** read from the stream, such that a subsequent fgetc() will
** return the same character again.
*/

int __backspace(FILE *f)
{
    backspace_called = TRUE;
    return 1;
}

/* END of Retargeted I/O */
#endif  // USE_SERIAL_PORT
```

```
#ifdef STANDALONE

/*
** Exception Signaling and Handling
** ================================
** The C library implementations of ferror() uses semihosting directly
** and must therefore be retargeted.  This is a minimal reimplementation.
** _sys_exit() is called after the user's main() function has exited.  The C library
** implementation uses semihosting to report to the debugger that the application
 has
** finished executing.
*/

int ferror(FILE *f)
{
    return EOF;
}

void _sys_exit(int return_code)
{
    while(1);
}

#endif // STANDALONE
```

Create the file `sp804_timer.c` containing the following code:

```
// -----------------------------------------------------------------
// SP804 Dual Timer
//
// Copyright (c) 2009-2017 Arm Limited (or its affiliates). All rights reserved.
// Use, modification and redistribution of this file is subject to your possession
 of a
// valid End User License Agreement for the Arm Product of which these examples are
 part of
// and your compliance with all applicable terms and conditions of such licence
 agreement.
// -----------------------------------------------------------------

#include "sp804_timer.h"

#define TIMER_SP804_CTRL_TIMEREN     (1 << 7)
#define TIMER_SP804_CTRL_TIMERMODE   (1 << 6)        // Bit 6:
#define TIMER_SP804_CTRL_INTENABLE   (1 << 5)
#define TIMER_SP804_CTRL_TIMERSIZE   (1 << 1)        // Bit 1: 0=16-bit, 1=32-bit
#define TIMER_SP804_CTRL_ONESHOT     (1 << 0)        // Bit 0: 0=wrapping, 1=one-
shot

#define TIMER_SP804_CTRL_PRESCALE_1  (0 << 2)        // clk/1
#define TIMER_SP804_CTRL_PRESCALE_4  (1 << 2)        // clk/4
#define TIMER_SP804_CTRL_PRESCALE_8  (2 << 2)        // clk/8

struct sp804_timer
{
        volatile uint32_t Time1Load;     // +0x00
  const volatile uint32_t Time1Value;    // +0x04 - RO
        volatile uint32_t Timer1Control; // +0x08
        volatile uint32_t Timer1IntClr;  // +0x0C - WO
  const volatile uint32_t Timer1RIS;     // +0x10 - RO
  const volatile uint32_t Timer1MIS;     // +0x14 - RO
        volatile uint32_t Timer1BGLoad;  // +0x18

        volatile uint32_t Time2Load;     // +0x20
        volatile uint32_t Time2Value;    // +0x24
        volatile uint8_t  Timer2Control; // +0x28
        volatile uint32_t Timer2IntClr;  // +0x2C - WO
  const volatile uint32_t Timer2RIS;     // +0x30 - RO
  const volatile uint32_t Timer2MIS;     // +0x34 - RO
```

```
        volatile uint32_t Timer2BGLoad;  // +0x38

  // Not including ID registers

};

// Instance of the dual timer, will be placed using the scatter file
struct sp804_timer* dual_timer;


// Set base address of timer
// address - virtual address of SP804 timer
void setTimerBaseAddress(uint64_t address)
{
  dual_timer = (struct sp804_timer*)address;
  return;
}


// Sets up the private timer
// load_value  - Initial value of timer
// auto_reload - Periodic (SP804_AUTORELOAD) or one shot (SP804_SINGLESHOT)
// interrupt   - Whether to generate an interrupt
void initTimer(uint32_t load_value, uint32_t auto_reload, uint32_t interrupt)
{
  uint32_t tmp = 0;

  dual_timer->Time1Load = load_value;

  // Fixed setting: 32-bit, no prescaling
  tmp = TIMER_SP804_CTRL_TIMERSIZE | TIMER_SP804_CTRL_PRESCALE_1 |
 TIMER_SP804_CTRL_TIMERMODE;

  // Settings from parameters: interrupt generation & reload
  tmp = tmp | interrupt | auto_reload;

  // Write control register
  dual_timer->Timer1Control = tmp;

  return;
}


// Starts the timer
void startTimer(void)
{
  uint32_t tmp;

  tmp = dual_timer->Timer1Control;
  tmp = tmp | TIMER_SP804_CTRL_TIMEREN;      // Set TimerEn (bit 7)
  dual_timer->Timer1Control = tmp;

  return;
}


// Stops the timer
void stopTimer(void)
{
  uint32_t tmp;

  tmp = dual_timer->Timer1Control;
  tmp = tmp & ~TIMER_SP804_CTRL_TIMEREN;     // Clear TimerEn (bit 7)
  dual_timer->Timer1Control = tmp;

  return;
}


// Returns the current timer count
uint32_t getTimerCount(void)
```

```
{
  return dual_timer->Time1Value;
}


void clearTimerIrq(void)
{
  // A write to this register, of any value, clears the interrupt
  dual_timer->Timer1IntClr = 1;
}


// ----------------------------------------------------------------
// End of sp804_timer.c
// ----------------------------------------------------------------
```

Create the file `sp804_timer.h` containing the following code:

```
// ----------------------------------------------------------------
// SP804 Dual Timer
// Header Filer
//
// Copyright (c) 2009-2017 Arm Limited (or its affiliates). All rights reserved.
// Use, modification and redistribution of this file is subject to your possession
 of a
// valid End User License Agreement for the Arm Product of which these examples are
 part of
// and your compliance with all applicable terms and conditions of such licence
 agreement.
// ----------------------------------------------------------------

#ifndef _SP804_TIMER_
#define _SP804_TIMER_

#include <stdint.h>

// Set base address of timer
// address - virtual address of SP804 timer
void setTimerBaseAddress(uint64_t address);


// Sets up the private timer
// load_value  - Initial value of timer
// auto_reload - Periodic (SP804_AUTORELOAD) or one shot (SP804_SINGLESHOT)
// interrupt   - Whether to generate an interrupt

#define SP804_AUTORELOAD   (0)
#define SP804_SINGLESHOT   (1)
#define SP804_GENERATE_IRQ (1 << 5)
#define SP804_NO_IRQ       (0)

void initTimer(uint32_t load_value, uint32_t auto_reload, uint32_t interrupt);


// Starts the timer
void startTimer(void);


// Stops the timer
void stopTimer(void);


// Returns the current timer count
uint32_t getTimerCount(void);


// Clears the timer interrupt
void clearTimerIrq(void);
```

```
#endif

// -------------------------------------------------------------
// End of sp804_timer.h
// -------------------------------------------------------------
```

Create the file `timer_interrupts.c` containing the following code:

```
/* Bare-metal example for Armv8-A Base FVP model */

/* Timer and interrupts */

/* Copyright (c) 2016 Arm Limited (or its affiliates). All rights reserved. */
/* Use, modification and redistribution of this file is subject to your possession
 of a     */
/* valid End User License Agreement for the Arm Product of which these examples are
 part of */
/* and your compliance with all applicable terms and conditions of such licence
 agreement.  */

#include <stdio.h>

#include "GICv3.h"
#include "GICv3_gicc.h"
#include "sp804_timer.h"

// LED Base address
#define LED_BASE (volatile unsigned int *)0x1C010008


void nudge_leds(void) // Move LEDs along
{
    static int state = 1;
    static int value = 1;

    if (state)
    {
        int max = (1 << 7);
        value <<= 1;
        if (value == max)
            state = 0;
    }
    else
    {
        value >>= 1;
        if (value == 1)
            state = 1;
    }

    *LED_BASE = value;  // Update LEDs hardware
}


// Initialize Timer 0 and Interrupt Controller
void init_timer(void)
{
    // Enable interrupts
    __asm("MSR DAIFClr, #0xF");
    setICC_IGRPEN1_EL1(igrpEnable);

    // Configure the SP804 timer to generate an interrupt
    setTimerBaseAddress(0x1C110000);
    initTimer(0x2000, SP804_AUTORELOAD, SP804_GENERATE_IRQ);
    startTimer();

    // The SP804 timer generates SPI INTID 34.  Enable
    // this ID, and route it to core 0.0.0.0 (this one!)
```

```
    SetSPIRoute(34, 0, gicdirouter_ModeSpecific);    // Route INTID 34 to 0.0.0.0
 (this core)
    SetSPIPriority(34, 0);                           // Set INTID 34 to priority to
 0
    ConfigureSPI(34, gicdicfgr_Level);               // Set INTID 34 as level-
sensitive
    EnableSPI(34);                                   // Enable INTID 34
}


// -------------------------------------------------------

void irqHandler(void)
{
  unsigned int ID;

  ID = getICC_IAR1(); // readIntAck();

  // Check for reserved IDs
  if ((1020 <= ID) && (ID <= 1023))
  {
      printf("irqHandler() - Reserved INTID %d\n\n", ID);
      return;
  }

  switch(ID)
  {
    case 34:
      // Dual-Timer 0 (SP804)
      printf("irqHandler() - External timer interrupt\n\n");
      nudge_leds();
      clearTimerIrq();
      break;

    default:
      // Unexpected ID value
      printf("irqHandler() - Unexpected INTID %d\n\n", ID);
      break;
  }

  // Write the End of Interrupt register to tell the GIC
  // we have finished handling the interrupt
  setICC_EOIR1(ID); // writeAliasedEOI(ID);
}

// -------------------------------------------------------

// Not actually used in this example, but provided for completeness

void fiqHandler(void)
{
  unsigned int ID;
  unsigned int aliased = 0;

  ID = getICC_IAR0(); // readIntAck();
  printf("fiqHandler() - Read %d from IAR0\n", ID);

  // Check for reserved IDs
  if ((1020 <= ID) && (ID <= 1023))
  {
    printf("fiqHandler() - Reserved INTID %d\n\n", ID);
    ID = getICC_IAR1(); // readAliasedIntAck();
    printf("fiqHandler() - Read %d from AIAR\n", ID);
    aliased = 1;

    // If still spurious then simply return
    if ((1020 <= ID) && (ID <= 1023))
        return;
  }

  switch(ID)
```

```
  {
    case 34:
      // Dual-Timer 0 (SP804)
      printf("fiqHandler() - External timer interrupt\n\n");
      clearTimerIrq();
      break;

    default:
      // Unexpected ID value
      printf("fiqHandler() - Unexpected INTID %d\n\n", ID);
      break;
  }

  // Write the End of Interrupt register to tell the GIC
  // we have finished handling the interrupt
  // NOTE: If the ID was read from the Aliased IAR, then
  // the aliased EOI register must be used
  if (aliased == 0)
    setICC_EOIR0(ID); // writeEOI(ID);
  else
    setICC_EOIR1(ID); // writeAliasedEOI(ID);
}
```

Create the file `uart.c` containing the following code:

```
/*
 * PL011 UART driver
 *
 * Copyright (c) 2005-2014 Arm Limited (or its affiliates). All rights reserved.
 * Use, modification and redistribution of this file is subject to your possession
 of a
 * valid End User License Agreement for the Arm Product of which these examples are
 part of
 * and your compliance with all applicable terms and conditions of such licence
 agreement.
 */

#include <stdio.h>

#include "uart.h"

/*
 * UART instance: will be placed using the scatter file
 */
static struct pl011_uart uart;

void UartInit(void)
{
    /*
     * ensure the UART is disabled
     */
    uart.UARTCR  = 0x0;

    /*
     * OK, now program this thing up
     */
    uart.UARTECR   = 0x0;  // Clear the receive status (i.e. error) register
    uart.UARTLCR_H = 0x0 | PL011_LCR_WORD_LENGTH_8 | PL011_LCR_FIFO_DISABLE | \
        PL011_LCR_ONE_STOP_BIT | PL011_LCR_PARITY_DISABLE | PL011_LCR_BREAK_DISABLE;

    uart.UARTIBRD = PL011_IBRD_DIV_38400;
    uart.UARTFBRD = PL011_FBRD_DIV_38400;

    /*
     * mask and clear all interrupts
     */
    uart.UARTIMSC = 0x0;
    uart.UARTICR  = PL011_ICR_CLR_ALL_IRQS;
```

```
    uart.UARTCR  = 0x0 | PL011_CR_UART_ENABLE | PL011_CR_TX_ENABLE |
 PL011_CR_RX_ENABLE;

    return;
}

void uart_putc_polled(char c)
{
    /* Wait for UART to become free */
    /* Note that FIFOs are not being used here */
    while (uart.UARTFR & PL011_FR_BUSY_FLAG);

    /* Write character and send it */
    uart.UARTDR = c;
}

char uart_getchar_polled(void)
{
    /* Wait for UART to become free */
    /* Note that FIFOs are not being used here */
    while (uart.UARTFR & PL011_FR_BUSY_FLAG);
    /* Read character received */
    return uart.UARTDR;
}
```

Create the file `uart.h` containing the following code:

```
/*
 * PL011 UART driver
 *
 * Copyright (c) 2005-2016 Arm Limited (or its affiliates). All rights reserved.
 * Use, modification and redistribution of this file is subject to your possession
 of a
 * valid End User License Agreement for the Arm Product of which these examples are
 part of
 * and your compliance with all applicable terms and conditions of such licence
 agreement.
 */

#ifndef uart_h
#define uart_h

/*
 * the layout of the UART device
 */
struct pl011_uart
{
        volatile unsigned int UARTDR;        // +0x00
        volatile unsigned int UARTECR;       // +0x04
  const volatile unsigned int unused0[4];    // +0x08 to +0x14 reserved
  const volatile unsigned int UARTFR;        // +0x18 - RO
  const volatile unsigned int unused1;       // +0x1C reserved
        volatile unsigned int UARTILPR;      // +0x20
        volatile unsigned int UARTIBRD;      // +0x24
        volatile unsigned int UARTFBRD;      // +0x28
        volatile unsigned int UARTLCR_H;     // +0x2C
        volatile unsigned int UARTCR;        // +0x30
        volatile unsigned int UARTIFLS;      // +0x34
        volatile unsigned int UARTIMSC;      // +0x38
  const volatile unsigned int UARTRIS;       // +0x3C - RO
  const volatile unsigned int UARTMIS;       // +0x40 - RO
        volatile unsigned int UARTICR;       // +0x44 - WO
        volatile unsigned int UARTDMACR;     // +0x48
};

/*
 * defines for control/status registers
```

```
 */
#define PL011_LCR_WORD_LENGTH_8   (0x60)
#define PL011_LCR_WORD_LENGTH_7   (0x40)
#define PL011_LCR_WORD_LENGTH_6   (0x20)
#define PL011_LCR_WORD_LENGTH_5   (0x00)

#define PL011_LCR_FIFO_ENABLE     (0x10)
#define PL011_LCR_FIFO_DISABLE    (0x00)

#define PL011_LCR_TWO_STOP_BITS   (0x08)
#define PL011_LCR_ONE_STOP_BIT    (0x00)

#define PL011_LCR_PARITY_ENABLE   (0x02)
#define PL011_LCR_PARITY_DISABLE  (0x00)

#define PL011_LCR_BREAK_ENABLE    (0x01)
#define PL011_LCR_BREAK_DISABLE   (0x00)

#define PL011_IBRD_DIV_38400      (0x27)
#define PL011_FBRD_DIV_38400      (0x09)

#define PL011_ICR_CLR_ALL_IRQS    (0x07FF)

#define PL011_FR_BUSY_FLAG        (0x08)
#define PL011_FR_RXFE_FLAG        (0x10)
#define PL011_FR_TXFF_FLAG        (0x20)
#define PL011_FR_RXFF_FLAG        (0x40)
#define PL011_FR_TXFE_FLAG        (0x80)

#define PL011_CR_UART_ENABLE      (0x01)

#define PL011_CR_TX_ENABLE        (0x0100)
#define PL011_CR_RX_ENABLE        (0x0200)


void UartInit(void);
void uart_putc_polled(char c);
char uart_getchar_polled(void);

#endif
```

Create the file `v8_aarch64.h` containing the following code:

```
/*
 *
 * Armv8-A AArch64 common helper functions
 *
 * Copyright (c) 2012-2016 Arm Limited (or its affiliates). All rights reserved.
 * Use, modification and redistribution of this file is subject to your possession
 of a
 * valid End User License Agreement for the Arm Product of which these examples are
 part of
 * and your compliance with all applicable terms and conditions of such licence
 agreement.
 */

#ifndef V8_AARCH64_H
#define V8_AARCH64_H

/*
 * Parameters for data barriers
 */
#define OSHLD   1
#define OSHST   2
#define OSH     3
#define NSHLD   5
#define NSHST   6
#define NSH     7
```

```
#define ISHLD   9
#define ISHST  10
#define ISH    11
#define LD     13
#define ST     14
#define SY     15

/*****************************************************************************/

/*
 * function prototypes
 */

/*
 * void InvalidateUDCaches(void)
 *   invalidates all Unified and Data Caches
 *
 * Inputs
 *   <none>
 *
 * Returns
 *   <nothing>
 *
 * Side Effects
 *   guarantees that all levels of cache will be invalidated before
 *   returning to caller
 */
void InvalidateUDCaches(void);

/*
 * unsigned long long EnableCachesEL1(void)
 *   enables I- and D- caches at EL1
 *
 * Inputs
 *   <none>
 *
 * Returns
 *   New value of SCTLR_EL1
 *
 * Side Effects
 *   context will be synchronised before returning to caller
 */
unsigned long long EnableCachesEL1(void);

/*
 * unsigned long long GetMIDR(void)
 *   returns the contents of MIDR_EL0
 *
 * Inputs
 *   <none>
 *
 * Returns
 *   MIDR_EL0
 */
unsigned long long GetMIDR(void);

/*
 * unsigned long long GetMPIDR(void)
 *   returns the contents of MPIDR_EL0
 *
 * Inputs
 *   <none>
 *
 * Returns
 *   MPIDR_EL0
 */
unsigned long long GetMPIDR(void);

/*
 * unsigned int GetCPUID(void)
 *   returns the Aff0 field of MPIDR_EL0
```

```
 *
 * Inputs
 *   <none>
 *
 * Returns
 *   MPIDR_EL0[7:0]
 */
unsigned int GetCPUID(void);

#endif
```

15. Overview of the Linker

The linker combines the contents of one or more object files with selected parts of one or more object libraries to produce executable images, partially linked object files, or shared object files.

Summary of the linker features

The linker has many features for linking input files to generate various types of output files.

The linker can:

- Link A32 and T32 code, or A64 code.
- Generate interworking veneers to switch between A32 and T32 states when required.
- Generate range extension veneers, where required, to extend the range of branch instructions.
- Automatically select the appropriate standard C or C++ library variants to link with, based on the build attributes of the objects it is linking.
- Position code and data at specific locations within the system memory map, using either a command-line option or a scatter file.
- Perform RW data compression to minimize ROM size.
- Eliminate unused sections to reduce the size of your output image.
- Control the generation of debug information in the output file.
- Generate a static callgraph and list the stack usage.
- Control the contents of the symbol table in output images.
- Show the sizes of code and data in the output.
- Build images suitable for all states of the Arm®v8-M Security Extension.


Note

Be aware of the following:

- Generated code might be different between two Arm Compiler for Embedded FuSa releases.
- For a feature release, there might be significant code generation differences.
- You cannot link A32 or T32 code with A64 code.


Note

The command-line option descriptions and related information in the *Arm Compiler for Embedded FuSa Reference Guide* describe all the features that Arm Compiler for Embedded FuSa supports. Any features not documented are not supported and are used at your own risk. You are responsible for making sure that any generated code using community features is operating correctly. For more information, see Support level definitions.

15.1 armlink command-line syntax

The `armlink` command can accept many input files together with options that determine how to process the files.

The command for invoking `armlink` is:

```
armlink <options> <input-file-list>
```

where:

`<options>`

`armlink` command-line options.

`<input-file-list>`

A space-separated list of objects, libraries, or symbol definitions (symdefs) files.

Related information

input-file-list linker option
Linker Command-line Options

15.2 What the linker does when constructing an executable image

`armlink` performs many operations, depending on the content of the input files and the command-line options you specify.

When you use the linker to construct an executable image, it:

- Resolves symbolic references between the input object files.
- Extracts object modules from libraries to satisfy otherwise unsatisfied symbolic references.
- Removes unused sections.
- Eliminates duplicate common section groups.
- Sorts input sections according to their attributes and names, and merges sections with similar attributes and names into contiguous chunks.
- Organizes object fragments into memory regions according to the grouping and placement information provided.
- Assigns addresses to relocatable values.
- Generates an executable image.

Related information

Elimination of unused sections
The structure of an Arm ELF image

15.3 What the linker can accept as input

`armlink` can accept one or more object files from toolchains that support Arm ELF.

Object files must be formatted as Arm® ELF. This format is described in:

- *ELF for the Arm Architecture (IHI 0044)*.
- *ELF for the Arm 64-bit Architecture (AArch64) (IHI 0056)*.

Optionally, the following files can be used as input to `armlink`:

- One or more libraries created by the librarian, `armar`.
- A symbol definitions file.
- A scatter file.
- A steering file.
- A Secure code import library when building a Non-secure image that needs to call a Secure image.
- A Secure code import library when building a Secure image that has to use the entry addresses in a previously generated import library.

Related information

Overview of the Arm Librarian on page 409
Security features supported in Arm Compiler for Embedded FuSa on page 285
--import_cmse_lib_in=filename
Access symbols in another image
Scatter-loading Features
Scatter File Syntax
Linker Steering File Command Reference
ELF for the Arm Architecture
ELF for the Arm 64-bit Architecture (AArch64)

15.4 What the linker outputs

`armlink` can create executable images and object files.

Output from `armlink` can be:

- An ELF executable image.
- A partially linked ELF object that can be used as input in a subsequent link step.
- A Secure code import library that is required by developers building a Non-secure image that needs to call a Secure image.


Note

You can also use `fromelf` to convert an ELF executable image to other file formats, or to display, process, and protect the content of an ELF executable image.

Related information

Security features supported in Arm Compiler for Embedded FuSa on page 285
Overview of the fromelf Image Converter on page 399
Partial linking model
Section placement with the linker
The structure of an Arm ELF image
--import_cmse_lib_out=filename

16. Getting Image Details

The linker provides options for getting information about the files it generates.

You can use the following options to get information about how your file is generated by the linker, and about the properties of the files:

--info
Displays information about various topics.

--map
Displays the image memory map, and contains the address and the size of each load region, execution region, and input section in the image, including linker-generated input sections. It also shows how RW data compression is applied.

--show_cmdline
Outputs the command-line used by the linker.

--symbols
Displays a list of each local and global symbol used in the link step, and its value.

--verbose
Displays detailed information about the link operation, including the objects that are included and the libraries that contain them.

--xref
Displays a list of all cross-references between input sections.

--xrefdbg
Displays a list of all cross-references between input debug sections.

The information can be written to a file using the `--list=<filename>` option.

16.1 Identifying the source of some link errors

The linker provides options to help you identify the source of some link errors.

Procedure

To identify the source of some link errors, use `--info inputs`.
For example, you can search the output to locate undefined references from library objects or multiply defined symbols caused by retargeting some library functions and not others. Search backwards from the end of this output to find and resolve link errors.

You can also use the `--verbose` option to output similar text with additional information on the linker operations.

Related information

Getting Image Details on page 390

--info=topic[,topic,...] (armlink)
--verbose (armlink)

16.2 Example of using the --info linker option

An example of the `--info` output.

To display the component sizes when linking enter:

```
armlink --info sizes ...
```

Here, `sizes` gives a list of the Code and data sizes for each input object and library member in the image. Using this option implies `--info sizes,totals`.

The following example shows the output in tabular format with the totals separated out for easy reading:

```
Image component sizes

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

        30         16          0          0          0          0   foo.o
        56         10        960          0       1024        372   startup_ARMCM7.o

    ----------------------------------------------------------------------
        88         26        992          0       5120        372   Object Totals
         0          0         32          0       4096          0   (incl.
 Generated)
         2          0          0          0          0          0   (incl. Padding)

    ----------------------------------------------------------------------

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Member
 Name

         8          0          0          0          0         68   __main.o
         0          0          0          0          0          0   __rtentry.o
        12          0          0          0          0          0   __rtentry2.o
         8          4          0          0          0          0   __rtentry5.o
        52          8          0          0          0          0   __scatter.o
        26          0          0          0          0          0   __scatter_copy.o
        28          0          0          0          0          0   __scatter_zi.o
        10          0          0          0          0         68   defsig_exit.o
        50          0          0          0          0         88   defsig_general.o
        80         58          0          0          0         76
 defsig_rtmem_inner.o
        14          0          0          0          0         80
 defsig_rtmem_outer.o
        52         38          0          0          0         76
 defsig_rtred_inner.o
        14          0          0          0          0         80
 defsig_rtred_outer.o
        18          0          0          0          0         80   exit.o
        76          0          0          0          0         88   fclose.o
       470          0          0          0          0         88   flsbuf.o
       236          4          0          0          0        128   fopen.o
        26          0          0          0          0         68   fputc.o
       248          6          0          0          0         84   fseek.o
        66          0          0          0          0         76   ftell.o
```

```
        94          0          0          0          0         80   h1_alloc.o
        52          0          0          0          0         68   h1_extend.o
        78          0          0          0          0         80   h1_free.o
        14          0          0          0          0         84   h1_init.o
        80          6          0          4          0         96   heapauxa.o
         4          0          0          0          0        136   hguard.o
         0          0          0          0          0          0   indicate_semi.o
       138          0          0          0          0        168   init_alloc.o
       312         46          0          0          0        112   initio.o
         2          0          0          0          0          0   libinit.o
         6          0          0          0          0          0   libinit2.o
        16          8          0          0          0          0   libinit4.o
         2          0          0          0          0          0   libshutdown.o
         6          0          0          0          0          0   libshutdown2.o
         0          0          0          0         96          0   libspace.o
         0          0          0          0          0          0
 maybetermalloc1.o
        44          4          0          0          0         84   puts.o
         8          4          0          0          0         68
 rt_errno_addr_intlibspace.o
         8          4          0          0          0         68
 rt_heap_descriptor_intlibspace.o
        78          0          0          0          0         80   rt_memclr_w.o
         2          0          0          0          0          0   rtexit.o
        10          0          0          0          0          0   rtexit2.o
        70          0          0          0          0         80   setvbuf.o
       240          6          0          0          0        156   stdio.o
         0          0          0         12        252          0   stdio_streams.o
        62          0          0          0          0         76   strlen.o
        12          4          0          0          0         68   sys_exit.o
       102          0          0          0          0        240   sys_io.o
         0          0         12          0          0          0   sys_io_names.o
        14          0          0          0          0         76   sys_wrch.o
         2          0          0          0          0         68   use_no_semi.o

    ----------------------------------------------------------------------
      2962        200         14         16        352       3036   Library Totals
        12          0          2          0          4          0   (incl. Padding)

    ----------------------------------------------------------------------

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Name

      2950        200         12         16        348       3036   c_wu.l

    ----------------------------------------------------------------------
      2962        200         14         16        352       3036   Library Totals

    ----------------------------------------------------------------------

==============================================================================


      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

      3050        226       1006         16       5472       1948   Grand Totals
      3050        226       1006         16       5472       1948   ELF Image Totals
      3050        226       1006         16          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                 4056 (   3.96kB)
    Total RW  Size (RW Data + ZI Data)              5488 (   5.36kB)
    Total ROM Size (Code + RO Data + RW Data)       4072 (   3.98kB)

==============================================================================
```

In this example:

Code (inc. data)

The number of bytes occupied by the code. In this image, there are 3050 bytes of code. This value includes 226 bytes of inline data (`inc. data`), for example, literal pools, and short strings.

RO Data

The number of bytes occupied by the RO data. This value is in addition to the inline data included in the `Code (inc. data)` column.

RW Data

The number of bytes occupied by the RW data.

ZI Data

The number of bytes occupied by the ZI data.

Debug

The number of bytes occupied by the debug data, for example, debug Input sections and the symbol and string table.

Object Totals

The number of bytes occupied by the objects when linked together to generate the image.

(incl. Generated)

`armlink` might generate image contents, for example, interworking veneers, and Input sections such as region tables. If the `Object Totals` row includes this type of data, it is shown in this row.

Combined across all of the object files (`foo.o` and `startup_ARMCM7.o`), the example shows that there are 992 bytes of RO data, of which 32 bytes are linker-generated RO data.


Note

If the scatter file contains `EMPTY` regions, the linker might generate ZI data. In the example, the 4096 bytes of ZI data labeled (`incl. Generated`) correspond to an `ARM_LIB_STACKHEAP` execution region used to set up the stack and heap in a scatter file as follows:

```
ARM_LIB_STACKHEAP +0x0 EMPTY 0x1000 {} ; 4KB stack + heap
```

Library Totals

The number of bytes occupied by the library members that have been extracted and added to the image as individual objects.

(incl. Padding)

If necessary, `armlink` inserts padding to force section alignment. If the `Object Totals` row includes this type of data, it is shown in the associated `(incl. Padding)` row. Similarly, if the `Library Totals` row includes this type of data, it is shown in its associated row.

In the example, there are 992 bytes of RO data in the object total, of which 0 bytes is linker-generated padding, and 14 bytes of RO data in the library total, with 2 bytes of padding.

`Grand Totals`

Shows the true size of the image. In the example, there are 5120 bytes of ZI data (in `Object Totals`) and 352 of ZI data (in `Library Totals`) giving a total of 5472 bytes.

`ELF Image Totals`

If you are using RW data compression (the default) to optimize ROM size, the size of the final image changes. This change is reflected in the output from `--info`. Compare the number of bytes under `Grand Totals` and `ELF Image Totals` to see the effect of compression.

In the example, RW data compression is not enabled. If data is compressed, the RW value changes.


Note

Not supported for AArch64 state.

`ROM Totals`

Shows the minimum size of ROM required to contain the image. This size does not include ZI data and debug information that is not stored in the ROM.

Related information

Getting Image Details on page 390
--info=topic[,topic,...] (armlink)

16.3 How to find where a symbol is placed when linking

To find where a symbol is placed when linking you must find the section that defines the symbol, and ensure that the linker has not removed the section.

About this task

You can find where a symbol is placed with the `--keep="<section_id>"` and `--symbols` options. For example, if `<object>(<section>)` is the section containing the symbol, enter:

```
armlink --cpu=8-A.32 --keep="<object>(<section>)" --symbols s.o --output=s.axf
```


Note

You can also run `fromelf -s` on the resultant image.

As an example, do the following:

Procedure

1. Create the file `s.c` containing the following source code:

```
long long array[10] __attribute__ ((section ("ARRAY")));

int main(void)
{
    return sizeof(array);
}
```

2. Compile the source:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c s.c -o s.o
```

3. Link the object `s.o`, keeping the `ARRAY` symbol and displaying the symbols:

```
armlink --cpu=8-A.32 --keep="s.o(ARRAY)" --map --symbols s.o --output=s.axf
```

4. Locate the `ARRAY` symbol in the output, for example:

```
...
Execution Region ER_RW (Base: 0x000083a8, Size: 0x00000028, Max: 0xffffffff,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x000083a8   0x00000028   Data   RW            4    ARRAY               s.o
```

```
...
Execution Region ER_RW (Base: 0x00008360, Size: 0x00000050, Max: 0xffffffff,
 ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00008360   0x00000050   Data   RW            3    ARRAY               s.o
```

   This shows that the array is placed in execution region ER_RW.

Related information

Using fromelf to find where a symbol is placed in an executable ELF image on page 406
--keep=section_id (armlink)
--map --no_map (armlink)
-o filename --output=filename (armlink)
-c compiler option
-march compiler option
-o compiler option
--target compiler option

17. SysV Dynamic Linking

Arm® Compiler for Embedded FuSa 6 supports the System V (SysV) linking model and can produce SysV shared objects and executables. The feature allows building programs for SysV-like platforms.


Note

Cortex®-M processors do not support dynamic linking.

17.1 Build a SysV shared object

To build SysV shared libraries, compile the code for position independence using the `-fsysv` and `-fpic` options. Compiling for position independence is required because a shared library can load to any suitable address in the memory map. The linker options that are required to build a SysV shared library are `--sysv`, `--shared`, and `--fpic`.

About this task

Build the shared library and then run `fromelf` to examine the contents.

Procedure

1. Create the file `lib.c` containing the following code:

```
__attribute__((visibility("default")))
int lib_func(int a)
{
    return 5 * a;
}
```

2. Build the library:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c -fsysv -fpic lib.c
armlink --sysv --shared --fpic lib.o -o lib.so
```

3. Run `fromelf` with the `--only` option to see that the function `lib_func()` has the visibility set to default and is present in the dynamic symbol table:

```
fromelf -s --only=.dynsym lib.so
...
** Section #2 '.dynsym' (SHT_DYNSYM) [SHF_ALLOC]
    Size   : 32 bytes (alignment 4)
    Address: 0x00000110
    String table #3 '.dynstr'
    Last local symbol no. 0

    Symbol table .dynsym (1 symbols, 0 local)

      #  Symbol Name                Value      Bind  Sec  Type  Vis  Size
    ================================================================================

      1  lib_func                   0x00000144   Gb    4  Code  De   0x1c
...
```

17.2 Build a SysV executable

To build a SysV executable with position independence compile with the `-fsysv` option. Compiling with position independence is not required by some SysV systems. For example, Arm Linux executables always execute from a fixed address of `0x8000`. However, other operating systems that are based on the SysV model might decide to have position independent executables.

Before you begin

Build the `lib.o` shared library as described in Build a SysV shared object.

Build the image and then run `fromelf` to examine the contents.

Procedure

1. Create the file `app.c` containing the following code:

```
#include <stdio.h>

int lib_func(int a);

int main(void)
{
    printf("Result: %d.\n", lib_func(3));
    return 0;
}
```

2. Build the main executable:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c -fsysv app.c
armlink --sysv app.o lib.so -o app.axf
```

   The reference to function `lib_func()` gets resolved by `lib.so`.

3. Run `fromelf` with the `--only` option to see that the resulting image contains a `DT_NEEDED` tag that indicates library `lib.so` is needed by the executable:

```
fromelf -y --only=.dynamic app.axf
...
** Section #9 '.dynamic' (SHT_DYNAMIC) [SHF_ALLOC + SHF_WRITE]
    Size   : 168 bytes (alignment 4)
    Address: 0x00012c9c
    String table #4 '.dynstr'

      #  Tag Name             Value
    ============================================================================

      0  DT_NEEDED               1 (lib.so)
      1  DT_HASH             33100 (0x0000814c)
      2  DT_STRTAB           33156 (0x00008184)
      3  DT_SYMTAB           33124 (0x00008164)
      4  DT_STRSZ               17
      5  DT_SYMENT              16
      6  DT_PLTRELSZ             8
      7  DT_PLTGOT           77124 (0x00012d44)
      8  DT_DEBUG                0 (0x00000000)
      9  DT_JMPREL           33176 (0x00008198)
     10  DT_PLTREL              17 (DT_REL)
     11  DT_NULL                 0
...
```

When executed, a platform-specific dynamic loader processes information in the dynamic array, loads `lib.so`, resolves relocations in all loaded files, and passes control to the main executable. The program then outputs:

```
Result: 15.
```

18. Overview of the fromelf Image Converter

The `fromelf` image conversion utility allows you to modify ELF image and object files, and to display information on those files.

`fromelf` allows you to:

- Process Arm ELF object and image files that the compiler, assembler, and linker generate.
- Process all ELF files in an archive that `armar` creates, and output the processed files into another archive if necessary.
- Convert ELF images into other formats for use by ROM tools or for direct loading into memory. The formats available are:
  - Plain binary.
  - Motorola 32-bit S-record. (AArch32 state only).
  - Intel Hex-32. (AArch32 state only).
  - Byte oriented (Verilog Memory Model) hexadecimal.
- Display information about the input file, for example, disassembly output or symbol listings, to either `stdout` or a text file. Disassembly is generated in `armasm` assembler syntax and not GNU assembler syntax. Therefore you cannot reassemble disassembled output with `armclang`.


Note

`armasm` does not support features of Arm®v8.4-A and later architectures, even those back-ported to Armv8.2-A and Armv8.3-A.


Note

If your image is produced without debug information, `fromelf` cannot:

- Translate the image into other file formats.
- Produce a meaningful disassembly listing.


Note

The command-line option descriptions and related information in the *Arm Compiler for Embedded FuSa Reference Guide* describe all the features that Arm Compiler for Embedded FuSa supports. Any features not documented are not supported and are used at your own risk. You are responsible for making sure that any generated code using community features is operating correctly. For more information, see Support level definitions.

18.1 fromelf execution modes

You can run `fromelf` in various execution modes.

The execution modes are:

- ELF mode (`--elf`), to resave a file as ELF.
- Text mode (`--text`, and others), to output information about an object or image file.
- Format conversion mode (`--bin`, `--m32`, `--i32`, `--vhx`).

Related information

--bin (fromelf)
--elf (fromelf)
--i32 (fromelf)
--m32 (fromelf)
--text (fromelf)
--vhx (fromelf)

18.2 Getting help on the fromelf command

Use the `--help` option to display a summary of the main command-line options. This option is the default if you do not specify any options or files.

Procedure

To display the help information, enter:

```
fromelf --help
```

Related information

fromelf command-line syntax on page 400
--help (fromelf)

18.3 fromelf command-line syntax

You can specify an ELF file or library of ELF files on the `fromelf` command-line.

Syntax

```
fromelf <options> <input_file>
```

`<options>`

`fromelf` command-line options.

<input_file>

The ELF file or library file to be processed. When some options are used, multiple input files can be specified.

Related information

fromelf Command-line Options

input_file (fromelf)

19. Using fromelf

Describes how to use the `fromelf` image converter provided with Arm® Compiler for Embedded FuSa.

19.1 General considerations when using fromelf

There are some changes that you cannot make to an image with `fromelf`.

When using `fromelf` you cannot:

- Change the image structure or addresses, other than altering the base address of Motorola S-record or Intel Hex output with the `--base` option.
- Change a scatter-loaded ELF image into a non scatter-loaded image in another format. Any structural or addressing information must be provided to the linker at link time.

Related information

--base [[object_file::]load_region_ID=]num (fromelf)
input_file (fromelf)

19.2 Examples of processing ELF files in an archive

Examples of how you can process all ELF files in an archive, or a subset of those files. The processed files together with any unprocessed files are output to another archive.

Examples

Consider an archive, `test.a`, containing the following ELF files:

```
bmw.o
bmw1.o
call_c_code.o
newtst.o
shapes.o
strmtst.o
```

Example of processing all files in the archive

This example removes all debug, comments, notes and symbols from all the files in the archive:

```
fromelf --elf --strip=all test.a -o strip_all/
```

The example also creates an output archive with the name `test.a` in the subdirectory `strip_all`

Example of processing a subset of files in the archive

To remove all debug, comments, notes and symbols from only the `shapes.o` and the `strmtst.o` files in the archive, enter:

```
fromelf --elf --strip=all test.a(s*.o) -o subset/
```

The example also creates an output archive with the name `test.a` in the subdirectory `subset`. The archive contains the processed files together with the remaining files that are unprocessed.

To process the `bmw.o`, `bmw1.o`, and `newtst.o` files in the archive, enter:

```
fromelf --elf --strip=all test.a(??w*) -o subset/
```

Example of displaying a disassembled version of files in an archive

To display the disassembled version of `call_c_code.o` in the archive, enter:

```
fromelf --disassemble test.a(c*)
```

Note

On Unix systems your shell typically requires the parentheses to be escaped with backslashes. Alternatively, enclose the complete section specifier in double quotes, for example:

```
--entry="8+startup.o(startupseg)"
```

Related information

--disassemble (fromelf)
--elf (fromelf)
input_file (fromelf)
--output=destination (fromelf)
--strip=option[,option,...] (fromelf)

19.3 Options to protect code in image files with fromelf

If you are delivering images to third parties, then you might want to protect the code they contain.

To help you to protect this code, fromelf provides the `--strip` option and the `--privacy` option. These options remove or obscure the symbol names in the image. The option that you choose depends on how much information you want to remove. The effect of these options is different for image files.

Restrictions

You must use `--elf` with these options. Because you have to use `--elf`, you must also use `--output`.

Effect of the --privacy and --strip options for protecting code in image files

Option	Effect
`fromelf --elf --privacy`	Removes the whole symbol table. Removes the `.comment` section name. This section is marked as `[Anonymous Section]` in the output from the `fromelf` option `--text`. Gives section names a default value. For example, changes code section names to `'.text'`.
`fromelf --elf --strip=symbols`	Removes the whole symbol table. Section names remain the same.
`fromelf --elf --strip=localsymbols`	Removes local and mapping symbols. Retains section names and build attributes.

Example

To produce a new ELF executable image with the complete symbol table removed and with the various section names changed, enter:

```
fromelf --elf --privacy --output=outfile.axf infile.axf
```

Related information

Options to protect code in object files with fromelf on page 404
fromelf command-line syntax on page 400
--elf (fromelf)
--output=destination (fromelf)
--privacy (fromelf)
--strip=option[,option,...] (fromelf)

19.4 Options to protect code in object files with fromelf

If you are delivering objects to third parties, then you might want to protect the code they contain.

To help you to protect this code, fromelf provides the `--strip` option and the `--privacy` option. These options remove or obscure the symbol names in the object. The option you choose depends on how much information you want to remove. The effect of these options is different for object files.

Restrictions

You must use `--elf` with these options. Because you have to use `--elf`, you must also use `--output`.

Effect of the --privacy and --strip options for protecting code in object files

Option	Local symbols	Section names	Mapping symbols	Build attributes
`fromelf --elf --privacy`	Removes those local symbols that can be removed without loss of functionality. Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.	Gives section names a default value. For example, changes code section names to `'.text'`	Present	Present
`fromelf --elf --strip=symbols`	Removes those local symbols that can be removed without loss of functionality. Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.	Section names remain the same	Present	Present
`fromelf --elf --strip=localsymbols`	Removes those local symbols that can be removed without loss of functionality. Symbols that cannot be removed, such as the targets for relocations, are kept. For these symbols, the names are removed. These are marked as `[Anonymous Symbol]` in the `fromelf --text` output.	Section names remain the same	Present	Present

Example

To produce a new ELF object with the complete symbol table removed and various section names changed, enter:

```
fromelf --elf --privacy --output=outfile.o infile.o
```

Related information

Options to protect code in image files with fromelf on page 403
fromelf command-line syntax on page 400
--elf (fromelf)

--output=destination (fromelf)
--privacy (fromelf)
--strip=option[,option,...] (fromelf)

19.5 Option to print specific details of ELF files

`fromelf` can extract information from ELF files. For example, ELF header and section information. Specify the information to extract using the `--emit` command-line option.


Note

You can specify some of the `--emit` options using the `--text` option.

Examples

To print the contents of the data sections of an ELF file, `infile.axf`, enter:

```
fromelf --emit=data infile.axf
```

To print relocation information and the dynamic section contents for the ELF file `infile2.axf`, enter:

```
fromelf --emit=relocation_tables,dynamic_segment infile2.axf
```

Related information

fromelf command-line syntax on page 400
--emit=option[,option,...] (fromelf)
--text (fromelf)

19.6 Using fromelf to find where a symbol is placed in an executable ELF image

You can find where a symbol is placed in an executable ELF image.

About this task

To find where a symbol is placed in an ELF image file, use the `--text -s -v` options to view the symbol table and detailed information on each segment and section header, for example:

The symbol table identifies the section where the symbol is placed.

Procedure

1. Create the file `s.c` containing the following source code:

```
long long arr[10] __attribute__ ((section ("ARRAY")));

int main()
{
    return sizeof(arr);
}
```

2. Compile the source:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c s.c -o s.o
```

3. Link the object `s.o` and keep the `ARRAY` symbol:

```
armlink --cpu=8-A.32 --keep=s.o(ARRAY) s.o --output=s.axf
```

4. Run the `fromelf` command to display the symbol table and detailed information on each segment and section header:

```
fromelf --text -s -v s.o
```

5. Locate the `arr` symbol in the `fromelf` output, for example:

```
...

** Section #24
Name         : .symtab
Type         : SHT_SYMTAB (0x00000002)
Flags        : None (0x00000000)
Addr         : 0x00000000
File Offset  : 868 (0x364)
Size         : 464 bytes (0x1d0)
Link         : Section 1 (.strtab)
Info         : Last local symbol no = 26
Alignment    : 4
Entry Size   : 16

Symbol table .symtab (28 symbols, 26 local)

  #  Symbol Name                 Value      Bind  Sec  Type  Vis  Size
============================================================================
 ...
 27  arr                         0x00000000   Gb    5  Data  De   0x50
 ...
```

   The `Sec` column shows the section where the stack is placed. In this example, section `5`.

6. Locate the section identified for the symbol in the `fromelf` output, for example:

```
...

====================================
** Section #5
    Name         : ARRAY
    Type         : SHT_PROGBITS (0x00000001)
    Flags        : SHF_ALLOC + SHF_WRITE (0x00000003)
    Addr         : 0x00000000
    File Offset  : 88 (0x58)
    Size         : 80 bytes (0x50)
    Link         : SHN_UNDEF
    Info         : 0
    Alignment    : 8
    Entry Size   : 0
    ====================================
    ...
```

   This shows that the symbols are placed in an ARRAY section.

Related information

--text (fromelf)

20. Overview of the Arm Librarian

The Arm Librarian, `armar`, enables you to collect and maintain sets of ELF object files in standard format `ar` libraries.

You can pass these libraries to the linker in place of several ELF object files.

With `armar` you can:

- Create new libraries.
- Add files to a library.
- Replace individual files in a library.
- Replace all files in a library with specified files in a single operation.
- Control the placement of files in a library.
- Display information about a specified library. For example, list all members in a library.

A timestamp is also associated with each file that is added or replaced in a library.


Note

When you create, add, or replace object files in a library, `armar` creates a symbol table by default. However, debug symbols are not included by default.

20.1 Considerations when working with library files

There are some considerations you must be aware of when working with library files.

Be aware of the following:

- A library differs from a shared object or dynamically linked library (DLL) in that:
  - Symbols are imported from a shared object or DLL.
  - Code or data for symbols is extracted from an archive into the file being linked.
- Linking with an object library file might not produce the same results as linking with all the object files collected into the object library file. This is because the linker processes the input list and libraries differently:
  - Each object file in the input list appears in the output unconditionally, although unused areas are eliminated if the `armlink` option `--remove` is specified.
  - A member of a library file is only included in the output if it is referred to by an object file or a previously processed library file.

  The linker recognizes a collection of ELF files stored in an `ar` format file as a library. The contents of each ELF file form a single member in the library.

Related information

--remove, --no_remove (armlink)

20.2 armar command-line syntax

The `armar` command has options to specify how to process files and libraries.

Syntax

```
armar <options> <archive> [<file_list>]
```

<options>

`armar` command-line options.

<archive>

The filename of the library. A library file must always be specified.

<file_list>

The list of files to be processed.

Related information

armar Command-line Options
archive (armar)
file_list (armar)

20.3 Option to get help on the armar command

Use the `--help` option to display a summary of the main command-line options.

This is the default if you do not specify any options or source files.

Example

To display the help information, enter:

```
armar --help
```

21. Overview of the armasm Legacy Assembler

The `armasm` legacy assembler supports instructions, directives, and user-defined macros.


Note

Because `armasm` is deprecated, some newer architectural features are not supported.

Supported features

`armasm` supports the following:

- Unified Assembly Language (UAL) for both A32 and T32 code.
- Assembly language for A64 code.
- Advanced SIMD instructions in A64, A32, and T32 code.
- Floating-point instructions in A64, A32, and T32 code.
- Directives in assembly source code.
- Processing of user-defined macros.
- `SDOT` and `UDOT` instructions that are an optional extension in Arm®v8.2-A and Armv8.3-A.

Unsupported architectural features

`armasm` does not support some architectural features, such as:

- Features of Armv8.4-A and later architectures, even those back-ported to Armv8.2-A and Armv8.3-A.
- Half-precision floating-point multiply with add or multiply with subtract arithmetic operations. These instructions are an optional extension in Armv8.2-A and Armv8.3-A, and a mandatory extension in Armv8.4-A and later. See `+fp16fml` in the -mcpu command-line option in the *Arm Compiler for Embedded FuSa Reference Guide*.
- AArch64 Crypto instructions (for SHA512, SHA3, SM3, SM4). See `+crypto` in the -mcpu command-line option in the *Arm Compiler for Embedded FuSa Reference Guide*.
- AArch64 Scalable Vector Extension (SVE) instructions. See `+sve` in the -mcpu command-line option in the *Arm Compiler for Embedded FuSa Reference Guide*.
- Armv8.1-M and later.
- Armv8-R AArch64 and later.

21.1 How the assembler works

`armasm` reads the assembly language source code twice before it outputs object code. Each read of the source code is called a pass.

This is because assembly language source code often contains forward references. A forward reference occurs when a label is used as an operand, for example as a branch target, earlier in the code than the definition of the label. The assembler cannot know the address of the forward reference label until it reads the definition of the label.

During each pass, the assembler performs different functions. In the first pass, the assembler:

- Checks the syntax of the instruction or directive. It faults if there is an error in the syntax, for example if a label is specified on a directive that does not accept one.
- Determines the size of the instruction and data being assembled and reserves space.
- Determines offsets of labels within sections.
- Creates a symbol table containing label definitions and their memory addresses.

In the second pass, the assembler:

- Faults if an undefined reference is specified in an instruction operand or directive.
- Encodes the instructions using the label offsets from pass 1, where applicable.
- Generates relocations.
- Generates debug information if requested.
- Outputs the object file.

Memory addresses of labels are determined and finalized in the first pass. Therefore, the assembly code must not change during the second pass. All instructions must be seen in both passes. Therefore you must not define a symbol after a `:DEF:` test for the symbol. The assembler faults if it sees code in pass 2 that was not seen in pass 1.

Line not seen in pass 1

The following example shows that `num EQU 42` is not seen in pass 1 but is seen in pass 2:

```
    AREA x,CODE
    [ :DEF: foo
num EQU 42
    ]
foo DCD num
    END
```

Assembling this code generates the error:

```
A1903E: Line not seen in first pass; cannot be assembled.
```

Line not seen in pass 2

The following example shows that `MOV r1,r2` is seen in pass 1 but not in pass 2:

```
    AREA x,CODE
    [ :LNOT: :DEF: foo
    MOV r1, r2
    ]
foo MOV r3, r4
    END
```

Assembling this code generates the error:

```
A1909E: Line not seen in second pass; cannot be assembled.
```

Related information

Directives that can be omitted in pass 2 of the assembler
Two pass assembler diagnostics
Instruction and directive relocations
--diag_error=tag[,tag,...]
--debug

22. Supporting reference information

The various features in Arm® Compiler for Embedded FuSa might have different levels of support, ranging from fully supported product features to community features.

22.1 Support level definitions

Arm® Compiler for Embedded FuSa 6 is built on Clang and LLVM technology. Therefore, it has more functionality than the set of product features described in the documentation.

Arm welcomes feedback regarding the use of all Arm Compiler for Embedded FuSa 6 features, and intends to support users to a level that is appropriate for that feature. You can contact support at https://developer.arm.com/support.

The following definitions clarify the levels of support and guarantees on functionality that are expected from these features.

Identification in the documentation

All features that are documented in the Arm Compiler for Embedded FuSa 6 documentation are product features, except where explicitly stated. The limitations of non-product features are explicitly stated.

Product features

Product features are suitable for use in a production environment. The functionality is well-tested, and is expected to be stable across feature and update releases.

- Arm intends to give advance notice of significant functionality changes to product features.
- If you have a support and maintenance contract, Arm provides full support for use of all product features.
- Arm welcomes feedback on product features.
- Any issues with product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler for Embedded FuSa.

In addition to fully supported product features, some product features are only alpha or beta quality.

Beta product features

Beta product features are implementation complete, but have not been sufficiently tested to be regarded as suitable for use in production environments.

Beta product features are identified with [BETA].

- Arm endeavors to document known limitations on beta product features.
- Beta product features are expected to eventually become product features in a future release of Arm Compiler for Embedded FuSa 6.

- Arm encourages the use of beta product features, and welcomes feedback on them.
- Any issues with beta product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler for Embedded FuSa.

Alpha product features

Alpha product features are not implementation complete, and are subject to change in future releases, therefore the stability level is lower than in beta product features.

Alpha product features are identified with [ALPHA].

- Arm endeavors to document known limitations of alpha product features.
- Arm encourages the use of alpha product features, and welcomes feedback on them.
- Any issues with alpha product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler for Embedded FuSa.

Community features

Arm Compiler for Embedded FuSa 6 is built on LLVM technology and preserves the functionality of that technology where possible. This means that there are additional features available in Arm Compiler for Embedded FuSa that are not listed in the documentation. These additional features are known as community features. For information on these community features, see the Clang Compiler User's Manual.

Where community features are referenced in the documentation, they are identified with [COMMUNITY].

- Arm makes no claims about the quality level or the degree of functionality of these features, except when explicitly stated in this documentation.
- Functionality might change significantly between feature releases.
- Arm makes no guarantees that community features are going to remain functional across update releases, although changes are expected to be unlikely.

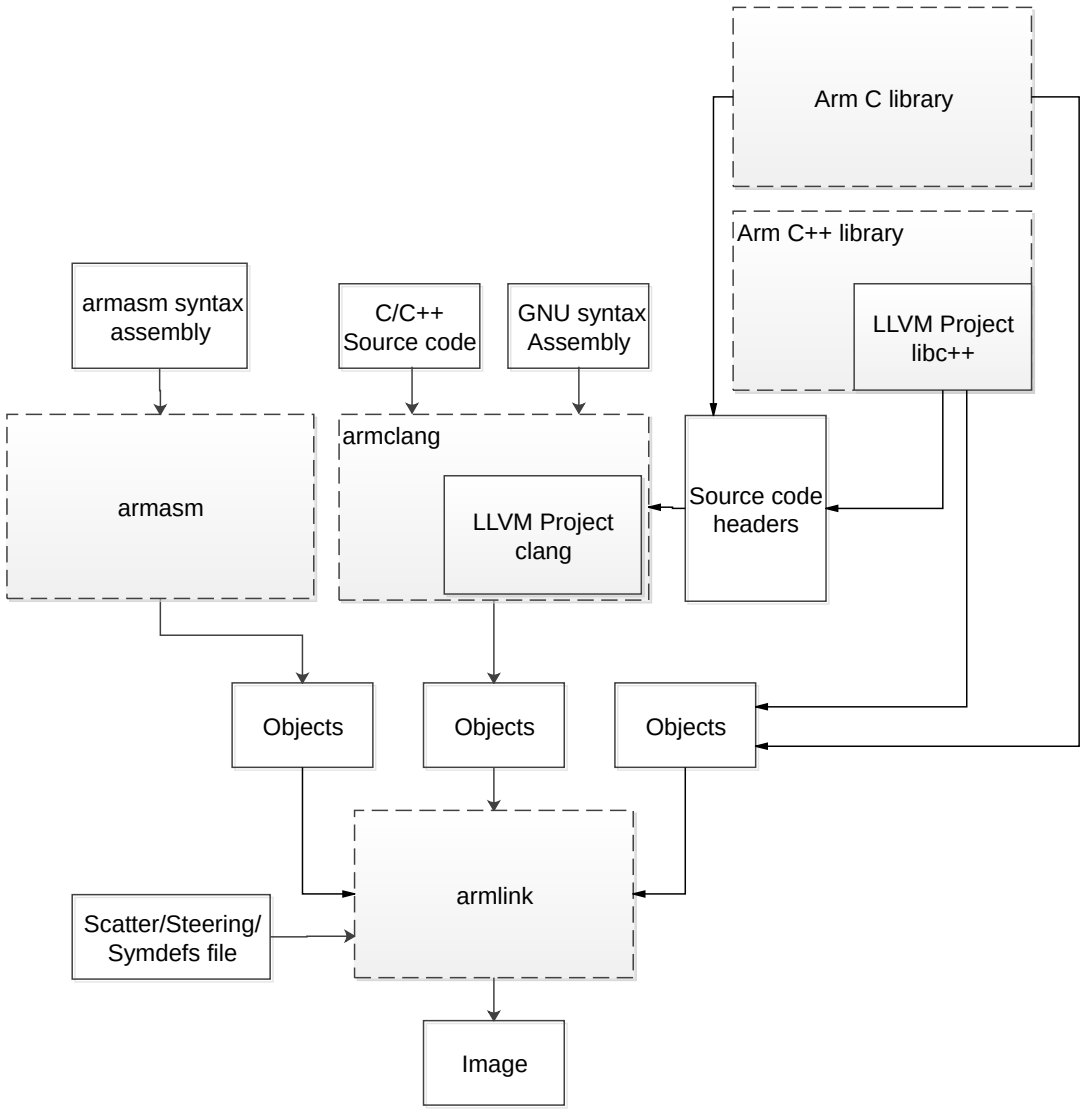
Some community features might become product features in the future, but Arm provides no roadmap for this. Arm is interested in understanding your use of these features, and welcomes feedback on them. Arm supports customers using these features on a best-effort basis, unless the features are unsupported. Arm accepts defect reports on these features, but does not guarantee that these issues are going to be fixed in future releases.

Guidance on use of community features

There are several factors to consider when assessing the likelihood of a community feature being functional:

- The following figure shows the structure of the Arm Compiler for Embedded FuSa 6 toolchain:

Figure 22-1: Integration boundaries in Arm Compiler for Embedded 6.



The dashed boxes are toolchain components, and any interaction between these components is an integration boundary. Community features that span an integration boundary might have significant limitations in functionality. The exception to this is if the interaction is codified in one of the standards supported by Arm Compiler for Embedded FuSa 6. See Application Binary Interface (ABI). Community features that do not span integration boundaries are more likely to work as expected.

- Features primarily used when targeting hosted environments such as Linux or BSD might have significant limitations, or might not be applicable, when targeting bare-metal environments.

- The Clang implementations of compiler features, particularly those that have been present for a long time in other toolchains, are likely to be mature. The functionality of new features, such as support for new language features, is likely to be less mature and therefore more likely to have limited functionality.

Deprecated features

A deprecated feature is one that Arm plans to remove from a future release of Arm Compiler for Embedded FuSa. We do not make any guarantee regarding the testing or maintenance of deprecated features. Therefore, we dos not recommend using a feature after it is deprecated.

For information on replacing deprecated features with supported features, see the Arm Compiler for Embedded FuSa documentation and Release Notes. Where appropriate, each Arm Compiler document includes notes for features that are deprecated, and also provides entries in the changes appendix of that document.

Unsupported features

With both the product and community feature categories, specific features and use cases are known not to function correctly, or are not intended for use with Arm Compiler for Embedded FuSa 6.

Limitations of product features are stated in the documentation. Arm cannot provide an exhaustive list of unsupported features or use cases for community features. The known limitations on community features are listed in Community features.

List of known unsupported features

The following is an incomplete list of unsupported features, and might change over time:

- The Clang option `-stdlib=libstdc++` is not supported.
- `-mabi=aapcs-soft` is not supported for A-profile targets in AArch64 state. The `aapcs-soft` ABI is defined only for Armv8-R AArch64 targets. For more information, see the *Soft-float* section of the Procedure Call Standard for the Arm 64-bit Architecture.
- `-mabi=aapcs-soft` is not supported for C++ source language modes.
- C++ static initialization of local variables is not thread-safe when linked against the standard C++ libraries. For thread-safety, you must provide your own implementation of thread-safe functions as described in Standard C++ library implementation definition.


Note

This restriction does not apply to the [ALPHA]-supported multithreaded C++ libraries.

- Use of C11 library features is unsupported.
- Any community feature that is exclusively related to non-Arm architectures is not supported.
- Except for Armv6-M, compilation for targets that implement architectures lower than Armv7 is not supported.

- The `long double` data type is not supported for AArch64 state because of limitations in the current Arm C library.
- C complex arithmetic is not supported, because of limitations in the current Arm C library.
- Complex numbers are defined in C++ as a template, `std::complex`. Arm Compiler for Embedded FuSa supports `std::complex` with the `float` and `double` types, but not the `long double` type because of limitations in the current Arm C library.

For C code that uses complex numbers, it is not sufficient to recompile with the C++ compiler to make that code work. How you can use complex numbers depends on whether or not you are building for Armv8-M targets.

- You must take care when mixing translation units that are compiled with and without the [COMMUNITY] `-fsigned-char` option, and that share interfaces or data structures.

The Arm ABI defines `char` as an unsigned byte, and this is the interpretation used by the C libraries supplied with the Arm compilation tools.

- There are limitations with the Control Flow Integrity (CFI) sanitizer implementation, `-fsanitize=cfi`, which requires Link-Time Optimization (LTO), `-flto`. The following are likely to occur:
  - When using features such as C++ I/O streams, the linker might report errors for a rejected local symbol, `L6654E`, or that a symbol is not preserved by the LTO code generation, `L6137E`.
  - The linker might report a diagnostic that a symbol has a size that extends outside of its containing section, `L6783E` or `L6784E`.

    Use the linker option `--diag_suppress 6783` or `--diag_suppress 6784` to suppress the diagnostic.

Alternatives to C complex numbers not being supported

If you are building for Armv8-M targets, consider using the free and open-source CMSIS-DSP library that includes a data type and library functions for complex number support in C. For more information about CMSIS-DSP and complex number support see the following sections of the CMSIS documentation:

- Complex Math Functions
- Complex Matrix Multiplication
- Complex FFT Functions

If you are not building for Armv8-M targets, consider modifying the affected part of your project to use the C++ standard library type `std::complex` instead.

22.2 Standards compliance in Arm Compiler for Embedded FuSa 6

Arm® Compiler for Embedded FuSa 6 conforms to the ISO C, ISO C++, ELF, and DWARF standards.

The level of compliance for each standard is:

ar

`armar` produces, and `armlink` consumes, UNIX-style object code archives. `armar` can list and extract most `ar`-format object code archives, and `armlink` can use an `ar`-format archive created by another archive utility providing it contains a symbol table member.

DWARF

The compiler generates DWARF 4 (DWARF Debugging Standard Version 4) debug tables with the `-g` option. The compiler can also generate DWARF 5 debug tables. Use DWARF 3 or DWARF 2 for backwards compatibility with legacy and third-party tools.

The linker can consume ELF format inputs containing DWARF 5, DWARF 4, DWARF 3, and DWARF 2 format debug tables.

The `fromelf` utility can consume ELF format inputs containing DWARF 4, DWARF 3, and DWARF 2 format debug tables. `fromelf` does not support DWARF 5.

This release provides a minimal implementation of DWARF 5 as follows:

- As a minimum, `armlink` correctly outputs DWARF 5.
- Although `fromelf -g` does not fail when processing DWARF 5 objects or images, `fromelf` cannot fully decode DWARF 5.
- `armlink` features `--callgraph`, `--info=stack`, and `--info=summarystack` process DWARF information to get the stack size for functions. It is possible that there might be DWARF 5-specific information that `armlink` cannot understand. We recommend compiling with DWARF 4 when using such features.

The legacy assembler `armasm` generates DWARF 3 debug tables with the `--debug` option. When assembling for AArch32, `armasm` can also generate DWARF 2 for backwards compatibility with legacy and third-party tools.

ISO C

The compiler accepts ISO C90, C99, and C11 source as input.

ISO C++

The compiler accepts ISO C++98, C++11, C++14, and C++17 source as input.

ELF

The toolchain produces relocatable and executable files in ELF format. The `fromelf` utility can translate ELF files into other formats.

Arm Compiler for Embedded FuSa and undefined behavior

The C and C++ standards consider any code that uses non-portable, erroneous program or data constructs as undefined behavior. Arm provides no information or guarantees about the behavior of Arm Compiler for Embedded FuSa when presented with a program that exhibits undefined behavior. That includes whether the compiler attempts to diagnose the undefined behavior.


Note

The `-fsanitize=undefined` command-line option is a [COMMUNITY] feature.

Related information

C++ implementation status in LLVM Clang

22.3 Compliance with the ABI for the Arm Architecture (Base Standard)

The ABI for the Arm Architecture (Base Standard) is a collection of standards. Some of these standards are open. Some are specific to the Arm architecture.

The *Application Binary Interface (ABI) for the Arm Architecture (Base Standard)* (BSABI) regulates the inter-operation of binary code and development tools in Arm® architecture-based execution environments, ranging from bare metal to major operating systems such as Arm Linux.

By conforming to this standard, objects produced by the toolchain can work together with object libraries from different producers.

The BSABI consists of a family of specifications including:

AADWARF64

DWARF for the Arm 64-bit Architecture (AArch64) with SVE support. This ABI uses the DWARF 3 standard to govern the exchange of debugging data between object producers and debuggers. It also gives additional rules on how to use DWARF 3, and how it is extended in ways specific to the 64-bit Arm architecture.

AADWARF

DWARF for the Arm Architecture. This ABI uses the DWARF 3 standard to govern the exchange of debugging data between object producers and debuggers.

AAELF64

ELF for the Arm 64-bit Architecture (AArch64). This specification provides the processor-specific definitions required by ELF for AArch64-based systems. It builds on the generic ELF standard to govern the exchange of linkable and executable files between producers and consumers.

AAELF

ELF for the Arm Architecture. Builds on the generic ELF standard to govern the exchange of linkable and executable files between producers and consumers.

AAPCS64

Procedure Call Standard for the Arm 64-bit Architecture (AArch64). Governs the exchange of control and data between functions at runtime. There is a variant of the AAPCS for each of the major execution environment types supported by the toolchain.

AAPCS64 describes a number of different supported data models. Arm Compiler for Embedded FuSa 6 implements the LP64 data model for AArch64 state.

AAPCS

Procedure Call Standard for the Arm 32-bit Architecture. Governs the exchange of control and data between functions at runtime. There is a variant of the AAPCS for each of the major execution environment types supported by the toolchain.

CLIBABI

C Library ABI for the Arm Architecture. Defines an ABI to the C library.

CPPABI64

C++ ABI for the Arm 64-bit Architecture. This specification builds on the generic C++ ABI (originally developed for IA-64) to govern interworking between independent C++ compilers.

CPPABI

C++ ABI for the Arm 32-bit Architecture. This specification builds on the generic C++ ABI to govern interworking between independent C++ compilers.

DBGOVL

Support for Debugging Overlaid Programs. Defines an extension to the ABI for the Arm Architecture to support debugging overlaid programs.

EHABI

Exception Handling ABI for the Arm Architecture. Defines both the language-independent and C++-specific aspects of how exceptions are thrown and handled.

RTABI

Run-time ABI for the Arm Architecture. Governs what independently produced objects can assume of their execution environments by way of floating-point and compiler helper-function support.

If you are upgrading from a previous toolchain release, ensure that you are using the most recent versions of the Arm specifications.

22.4 GCC compatibility provided by Arm Compiler for Embedded FuSa 6

The compiler in Arm® Compiler for Embedded FuSa 6 is based on Clang and LLVM technology. As such, it provides a high degree of compatibility with GCC.

Arm Compiler for Embedded FuSa 6 can build most of the C code that is written to be built with GCC. However, Arm Compiler for Embedded FuSa is not 100% source compatible in all cases. Specifically, Arm Compiler for Embedded FuSa does not aim to be bug-compatible with GCC. That is, Arm Compiler for Embedded FuSa does not replicate GCC bugs.

22.5 Locale support in Arm Compiler for Embedded FuSa 6

Summarizes the locales supported by Arm® Compiler for Embedded FuSa 6.

Arm Compiler for Embedded FuSa provides full support only for the English locale.

Arm Compiler for Embedded FuSa provides support for multibyte characters, for example Japanese characters, within comments in UTF-8 encoded files. This includes:

- `/* */` comments in C source files, C++ source files, and GNU-syntax assembly files.
- `//` comments in C source files, C++ source files, and GNU-syntax assembly files.
- `@` comments in GNU-syntax assembly files, for Arm architectures.
- `;` comments in `armasm`-syntax assembly source files and `armlink` scatter files.

Note

There is no support for Shift-Japanese Industrial Standard (Shift-JIS) encoded files.

22.6 Toolchain environment variables

Arm® Compiler for Embedded FuSa does not require environment variables to be set. However, there are situations where you might want to set environment variables.

The environment variables that the toolchain uses are described in the following table.

Where an environment variable is identified as GCC compatible, the GCC documentation provides full information about that environment variable. See https://gcc.gnu.org/onlinedocs/gcc/Environment-Variables.html at https://gcc.gnu.org.

To set an environment variable on a Windows machine:

1. Open the **System** settings from the Control Panel.
2. Click **Advanced system settings** to display the System Properties dialog box, then click **Environment Variables...**.
3. Create a new user variable for the required environment variable.

To set an environment variable on a Linux machine, open a `bash` shell and use the `export` command. For example:

```
export ARMCOMPILER6_CLANGOPT="-mabi=aapcs-soft -mtune=cortex-a57"
```

Table 22-1: Environment variables used by the toolchain

Environment variable	Setting
`ARMCOMPILER6_ASMOPT`	An optional environment variable to define additional assembler options that are to be used outside your regular makefile. The options listed appear before any options specified for the `armasm` command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.
`ARMCOMPILER6_CLANGOPT`	An optional environment variable to define additional `armclang` options that are to be used outside your regular makefile. The options listed appear before any options specified for the `armclang` command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.
`ARMCOMPILER6_FROMELFOPT`	An optional environment variable to define additional `fromelf` image converter options that are to be used outside your regular makefile. The options listed appear before any options specified for the `fromelf` command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.
`ARMCOMPILER6_LINKOPT`	An optional environment variable to define additional linker options that are to be used outside your regular makefile. The options listed appear before any options specified for the `armlink` command in the makefile. Therefore, any options specified in the makefile might override the options listed in this environment variable.
`ARMROOT`	Your installation directory root, `<install_directory>`.
`C_INCLUDE_PATH`	GCC-compatible environment variable. Adds the specified directories to the list of places that are searched to find included C files.
`COMPILER_PATH`	GCC-compatible environment variable. Adds the specified directories to the list of places that are searched to find subprograms.

Environment variable	Setting
`CPATH`	GCC-compatible environment variable. Adds the specified directories to the list of places that are searched to find included files regardless of the source language.
`CPLUS_INCLUDE_PATH`	GCC-compatible environment variable. Adds the specified directories to the list of places that are searched to find included C++ files.
`TMP`	Used on Windows platforms to specify the directory to be used for temporary files.
`TMPDIR`	Used on Red Hat Linux platforms to specify the directory to be used for temporary files.

22.7 Clang and LLVM documentation

Arm® Compiler for Embedded FuSa is based on Clang and LLVM compiler technology.

The Arm Compiler for Embedded FuSa documentation describes features that are specific to, and supported by, Arm Compiler for Embedded FuSa. Any features specific to Arm Compiler for Embedded FuSa that are not documented are not supported and are used at your own risk. Although open-source Clang features that Arm does not document are available, they are not supported by Arm and are used at your own risk. You are responsible for making sure that any generated code using unsupported or community features is operating correctly. For more information, see Support level definitions.

The https://releases.llvm.org/18.1.0/tools/clang/docs/UsersManual.html, available from the LLVM Compiler Infrastructure Project web site https://llvm.org, provides open-source documentation for Clang.

See the `third_party_licenses.txt` file in your installation for details of open-source software projects used.


Note

Although Arm Compiler for Embedded FuSa 6 is based on Clang and LLVM technology, it:

- Is not based on the same revision as any specific release of the open-source version of Clang or LLVM.
- Can contain changes introduced by Arm which are not included in the open-source version.

  The `third_party_licenses.txt` file includes GitHub links for the specific revisions in the open-source project which are relevant to the particular version of Arm Compiler for Embedded FuSa.

22.8 Extensions that are considered qualified features within the scope of functional safety certification

Certain language extensions have undergone successful language conformance testing for the C and C++ language standards supported by Arm® Compiler for Embedded FuSa 6.22LTS, and are included within the scope of functional safety certification. These extensions are considered to be qualified features.

Documented features

In general, features described in the Arm Compiler for Embedded FuSa 6.22LTS documentation as product features are supported by Arm Compiler for Embedded FuSa 6.22LTS and are considered to be within the scope of functional safety certification.

Although the product documentation does not describe language extensions by name, descriptions of product features might include the use of all or part of a language extension. This implies that the use of a language extension that is included in the description of a product feature is considered to be within the scope of functional safety certification.

For example, the following product features describe the use of language extensions. The described use cases are considered qualified features within the scope of functional safety certification because they are explicitly included in the product documentation:

Table 22-2: Documented product features and language extensions

Product feature	Extension used in documentation	Explanation
`$Sub$$` and `$Super$$` to patch symbol definitions	`dollar-in-identifier-extension`	The Use of $Super$$ and $Sub$$ to patch symbol definitions section of the *User Guide* describes a product feature that requires the use of the dollar signs `$` in identifiers. Because the product feature is a qualified feature and is within the scope of functional safety certification, and the extension `dollar-in-identifier-extension` relates wholly to the use of dollar signs in identifiers, the extension is also considered to be a qualified feature.
Inline assembly	`language-extension-token`	The Inline assembly with Arm Compiler for Embedded FuSa 6 section of the *Migration and Compatibility Guide* states that the `asm` keyword can be used in certain circumstances for writing inline assembly statements. The inline assembly product feature is a qualified feature and is within the scope of functional safety certification. The `asm` keyword is part of the extension `language-extension-token`. However, there are other keywords such as `typeof` that are part of this extension but are not mentioned in the product documentation. Therefore, only the `asm` keyword from the extension `language-extension-token` is considered to be a qualified feature as part of the inline assembler product feature.

Features from supported C and C++ standards

Arm Compiler for Embedded FuSa 6.22LTS supports C++ language standard up to and including ISO C++17 and C standards up to and including ISO C99. Extensions from these supported C and C++ standards that have been backported to earlier supported standards, or those which facilitate source code re-use between C and C++, are considered qualified features within the scope of functional safety certification.

For example, because the following extensions are all features from ISO C99, they are considered qualified in features:

- `c99-designator`
- `c99-extensions`
- `vla-extension`

Related information

Extensions that are outside the scope of functional safety certification on page 426

22.9 Extensions that are outside the scope of functional safety certification

Any extensions that are not explicitly covered by Extensions that are considered qualified features within the scope of functional safety certification are not included in scope of functional safety certification and are therefore not considered qualified features.

However, because the compiler is built on open-source Clang/LLVM technology, certain extensions have undergone sufficient testing from the open-source community that you can justify their use based on the results of such testing.

The following table provides references to the open-source Clang/LLVM tests for some example language extensions that have been tested by the open-source community. You can use these tests as evidence to help justify the use of these extensions as part of the qualification process for safety-related projects built using Arm® Compiler for Embedded FuSa 6.22LTS:

Table 22-3: Associated open-source Clang/LLVM tests for language extensions

Extension	Associated open-source Clang/LLVM tests
`c++20-designator`	• CodeGenCXX/designated-init.cpp • Parser/cxx2a-designated-init.cpp • SemaCXX/designated-initializers.cpp
`extra-semi`	• SemaCXX/extra-semi.cpp • Parser/cxx-extra-semi.cpp • Parser/extra-semi-resulting-in-nullstmt-in-init-statement.cpp • Parser/extra-semi-resulting-in-nullstmt.cpp

Extension	Associated open-source Clang/LLVM tests
`gnu-anonymous-struct`	• CodeGenCXX/anonymous-union-member-initializer.cpp • CodeGenCXX/cxx1y-initializer-aggregate.cpp • CodeGenCXX/debug-info-anon-union-vars.cpp • CodeGenCXX/member-init-anon-union.cpp • SemaCXX/anonymous-struct.cpp • SemaCXX/anonymous-union.cpp
`gnu-case-range`	• CodeGen/switch.c • Sema/switch.c • SemaCXX/gnu-case-ranges.cpp
`gnu-conditional-ommitted-operand`	• CodeGen/conditional-gnu-ext.c • CodeGenCXX/conditional-gnu-ext.cpp • Sema/const-eval.c
`gnu-designator`	• CodeGen/init.c • Parser/designator.c • Sema/designated-initializers.c
`gnu-empty-initializer`	• CodeGen/const-init.c • CodeGen/designated-initializers.c • Sema/array-init.c • Sema/flexible-array-init.c • Sema/init.c
`gnu-folding-constant`	• Sema/const-eval-64.c • Sema/const-eval.c • SemaCXX/constant-expression-cxx11.cpp • SemaCXX/constant-expression.cpp • SemaCXX/i-c-e-cxx.cpp
`gnu-statement-expression`	• CodeGen/2003-08-21-StmtExpr.c • CodeGen/exprs.c • CodeGenCXX/stmtexpr.cpp • Sema/stmtexprs.c • SemaCXX/statements.cpp
`gnu-zero-variadic-macro-arguments`	• Preprocessor/macro_expand.c • Preprocessor/macro_fn.c • Preprocessor/macro_fn_comma_swallow.c • Preprocessor/macro_fn_comma_swallow2.c • Preprocessor/macro_fn_varargs_named.c • Preprocessor/macro_paste_commaext.c • Preprocessor/macro_vaopt_expand.cpp

22.10 typinfo.s example source code

The `typinfo.s` source code used in the example for avoiding Run-Time Type Information (RTTI).

See the example in Avoid linking in Run-Time Type Information.

```
    .section unused_rtti, "aw", %nobits
    .weak _ZTIDh
    .weak _ZTIDi
    .weak _ZTIDn
    .weak _ZTIDs
    .weak _ZTIN10__cxxabiv116__enum_type_infoE
    .weak _ZTIN10__cxxabiv116__shim_type_infoE
    .weak _ZTIN10__cxxabiv117__array_type_infoE
    .weak _ZTIN10__cxxabiv117__class_type_infoE
    .weak _ZTIN10__cxxabiv117__pbase_type_infoE
    .weak _ZTIN10__cxxabiv119__pointer_type_infoE
    .weak _ZTIN10__cxxabiv120__function_type_infoE
    .weak _ZTIN10__cxxabiv120__si_class_type_infoE
    .weak _ZTIN10__cxxabiv121__vmi_class_type_infoE
    .weak _ZTIN10__cxxabiv123__fundamental_type_infoE
    .weak _ZTIN10__cxxabiv129__pointer_to_member_type_infoE
    .weak _ZTIPDh
    .weak _ZTIPDi
    .weak _ZTIPDn
    .weak _ZTIPDs
    .weak _ZTIPKDh
    .weak _ZTIPKDi
    .weak _ZTIPKDn
    .weak _ZTIPKDs
    .weak _ZTIPKa
    .weak _ZTIPKb
    .weak _ZTIPKc
    .weak _ZTIPKd
    .weak _ZTIPKe
    .weak _ZTIPKf
    .weak _ZTIPKg
    .weak _ZTIPKh
    .weak _ZTIPKi
    .weak _ZTIPKj
    .weak _ZTIPKl
    .weak _ZTIPKm
    .weak _ZTIPKn
    .weak _ZTIPKo
    .weak _ZTIPKs
    .weak _ZTIPKt
    .weak _ZTIPKv
    .weak _ZTIPKw
    .weak _ZTIPKx
    .weak _ZTIPKy
    .weak _ZTIPa
    .weak _ZTIPb
    .weak _ZTIPc
    .weak _ZTIPd
    .weak _ZTIPe
    .weak _ZTIPf
    .weak _ZTIPg
    .weak _ZTIPh
    .weak _ZTIPi
    .weak _ZTIPj
    .weak _ZTIPl
    .weak _ZTIPm
    .weak _ZTIPn
    .weak _ZTIPo
    .weak _ZTIPs
    .weak _ZTIPt
    .weak _ZTIPv
```

```
    .weak _ZTIPw
    .weak _ZTIPx
    .weak _ZTIPy
    .weak _ZTIa
    .weak _ZTIb
    .weak _ZTIc
    .weak _ZTId
    .weak _ZTIe
    .weak _ZTIf
    .weak _ZTIg
    .weak _ZTIh
    .weak _ZTIi
    .weak _ZTIj
    .weak _ZTIl
    .weak _ZTIm
    .weak _ZTIn
    .weak _ZTIo
    .weak _ZTIs
    .weak _ZTIt
    .weak _ZTIv
    .weak _ZTIw
    .weak _ZTIx
    .weak _ZTIy
    .weak _ZTSDh
    .weak _ZTSDi
    .weak _ZTSDn
    .weak _ZTSDs
    .weak _ZTSN10__cxxabiv116__enum_type_infoE
    .weak _ZTSN10__cxxabiv116__shim_type_infoE
    .weak _ZTSN10__cxxabiv117__array_type_infoE
    .weak _ZTSN10__cxxabiv117__class_type_infoE
    .weak _ZTSN10__cxxabiv117__pbase_type_infoE
    .weak _ZTSN10__cxxabiv119__pointer_type_infoE
    .weak _ZTSN10__cxxabiv120__function_type_infoE
    .weak _ZTSN10__cxxabiv120__si_class_type_infoE
    .weak _ZTSN10__cxxabiv121__vmi_class_type_infoE
    .weak _ZTSN10__cxxabiv123__fundamental_type_infoE
    .weak _ZTSN10__cxxabiv129__pointer_to_member_type_infoE
    .weak _ZTSPDh
    .weak _ZTSPDi
    .weak _ZTSPDn
    .weak _ZTSPDs
    .weak _ZTSPKDh
    .weak _ZTSPKDi
    .weak _ZTSPKDn
    .weak _ZTSPKDs
    .weak _ZTSPKa
    .weak _ZTSPKb
    .weak _ZTSPKc
    .weak _ZTSPKd
    .weak _ZTSPKe
    .weak _ZTSPKf
    .weak _ZTSPKg
    .weak _ZTSPKh
    .weak _ZTSPKi
    .weak _ZTSPKj
    .weak _ZTSPKl
    .weak _ZTSPKm
    .weak _ZTSPKn
    .weak _ZTSPKo
    .weak _ZTSPKs
    .weak _ZTSPKt
    .weak _ZTSPKv
    .weak _ZTSPKw
    .weak _ZTSPKx
    .weak _ZTSPKy
    .weak _ZTSPa
    .weak _ZTSPb
    .weak _ZTSPc
    .weak _ZTSPd
    .weak _ZTSPe
```

```
    .weak _ZTSPf
    .weak _ZTSPg
    .weak _ZTSPh
    .weak _ZTSPi
    .weak _ZTSPj
    .weak _ZTSPl
    .weak _ZTSPm
    .weak _ZTSPn
    .weak _ZTSPo
    .weak _ZTSPs
    .weak _ZTSPt
    .weak _ZTSPv
    .weak _ZTSPw
    .weak _ZTSPx
    .weak _ZTSPy
    .weak _ZTSa
    .weak _ZTSb
    .weak _ZTSc
    .weak _ZTSd
    .weak _ZTSe
    .weak _ZTSf
    .weak _ZTSg
    .weak _ZTSh
    .weak _ZTSi
    .weak _ZTSj
    .weak _ZTSl
    .weak _ZTSm
    .weak _ZTSn
    .weak _ZTSo
    .weak _ZTSs
    .weak _ZTSt
    .weak _ZTSv
    .weak _ZTSw
    .weak _ZTSx
    .weak _ZTSy
    .weak _ZTVN10__cxxabiv116__enum_type_infoE
    .weak _ZTVN10__cxxabiv116__shim_type_infoE
    .weak _ZTVN10__cxxabiv117__array_type_infoE
    .weak _ZTVN10__cxxabiv117__class_type_infoE
    .weak _ZTVN10__cxxabiv117__pbase_type_infoE
    .weak _ZTVN10__cxxabiv119__pointer_type_infoE
    .weak _ZTVN10__cxxabiv120__function_type_infoE
    .weak _ZTVN10__cxxabiv120__si_class_type_infoE
    .weak _ZTVN10__cxxabiv121__vmi_class_type_infoE
    .weak _ZTVN10__cxxabiv123__fundamental_type_infoE
    .weak _ZTVN10__cxxabiv129__pointer_to_member_type_infoE
_ZTIDh:
_ZTIDi:
_ZTIDn:
_ZTIDs:
_ZTIN10__cxxabiv116__enum_type_infoE:
_ZTIN10__cxxabiv116__shim_type_infoE:
_ZTIN10__cxxabiv117__array_type_infoE:
_ZTIN10__cxxabiv117__class_type_infoE:
_ZTIN10__cxxabiv117__pbase_type_infoE:
_ZTIN10__cxxabiv119__pointer_type_infoE:
_ZTIN10__cxxabiv120__function_type_infoE:
_ZTIN10__cxxabiv120__si_class_type_infoE:
_ZTIN10__cxxabiv121__vmi_class_type_infoE:
_ZTIN10__cxxabiv123__fundamental_type_infoE:
_ZTIN10__cxxabiv129__pointer_to_member_type_infoE:
_ZTIPDh:
_ZTIPDi:
_ZTIPDn:
_ZTIPDs:
_ZTIPKDh:
_ZTIPKDi:
_ZTIPKDn:
_ZTIPKDs:
_ZTIPKa:
_ZTIPKb:
```

```
_ZTIPKc:
_ZTIPKd:
_ZTIPKe:
_ZTIPKf:
_ZTIPKg:
_ZTIPKh:
_ZTIPKi:
_ZTIPKj:
_ZTIPKl:
_ZTIPKm:
_ZTIPKn:
_ZTIPKo:
_ZTIPKs:
_ZTIPKt:
_ZTIPKv:
_ZTIPKw:
_ZTIPKx:
_ZTIPKy:
_ZTIPa:
_ZTIPb:
_ZTIPc:
_ZTIPd:
_ZTIPe:
_ZTIPf:
_ZTIPg:
_ZTIPh:
_ZTIPi:
_ZTIPj:
_ZTIPl:
_ZTIPm:
_ZTIPn:
_ZTIPo:
_ZTIPs:
_ZTIPt:
_ZTIPv:
_ZTIPw:
_ZTIPx:
_ZTIPy:
_ZTIa:
_ZTIb:
_ZTIc:
_ZTId:
_ZTIe:
_ZTIf:
_ZTIg:
_ZTIh:
_ZTIi:
_ZTIj:
_ZTIl:
_ZTIm:
_ZTIn:
_ZTIo:
_ZTIs:
_ZTIt:
_ZTIv:
_ZTIw:
_ZTIx:
_ZTIy:
_ZTSDh:
_ZTSDi:
_ZTSDn:
_ZTSDs:
_ZTSN10__cxxabiv116__enum_type_infoE:
_ZTSN10__cxxabiv116__shim_type_infoE:
_ZTSN10__cxxabiv117__array_type_infoE:
_ZTSN10__cxxabiv117__class_type_infoE:
_ZTSN10__cxxabiv117__pbase_type_infoE:
_ZTSN10__cxxabiv119__pointer_type_infoE:
_ZTSN10__cxxabiv120__function_type_infoE:
_ZTSN10__cxxabiv120__si_class_type_infoE:
_ZTSN10__cxxabiv121__vmi_class_type_infoE:
```

```
_ZTSN10__cxxabiv123__fundamental_type_infoE:
_ZTSN10__cxxabiv129__pointer_to_member_type_infoE:
_ZTSPDh:
_ZTSPDi:
_ZTSPDn:
_ZTSPDs:
_ZTSPKDh:
_ZTSPKDi:
_ZTSPKDn:
_ZTSPKDs:
_ZTSPKa:
_ZTSPKb:
_ZTSPKc:
_ZTSPKd:
_ZTSPKe:
_ZTSPKf:
_ZTSPKg:
_ZTSPKh:
_ZTSPKi:
_ZTSPKj:
_ZTSPKl:
_ZTSPKm:
_ZTSPKn:
_ZTSPKo:
_ZTSPKs:
_ZTSPKt:
_ZTSPKv:
_ZTSPKw:
_ZTSPKx:
_ZTSPKy:
_ZTSPa:
_ZTSPb:
_ZTSPc:
_ZTSPd:
_ZTSPe:
_ZTSPf:
_ZTSPg:
_ZTSPh:
_ZTSPi:
_ZTSPj:
_ZTSPl:
_ZTSPm:
_ZTSPn:
_ZTSPo:
_ZTSPs:
_ZTSPt:
_ZTSPv:
_ZTSPw:
_ZTSPx:
_ZTSPy:
_ZTSa:
_ZTSb:
_ZTSc:
_ZTSd:
_ZTSe:
_ZTSf:
_ZTSg:
_ZTSh:
_ZTSi:
_ZTSj:
_ZTSl:
_ZTSm:
_ZTSn:
_ZTSo:
_ZTSs:
_ZTSt:
_ZTSv:
_ZTSw:
_ZTSx:
_ZTSy:
_ZTVN10__cxxabiv116__enum_type_infoE:
```

```
_ZTVN10__cxxabiv116__shim_type_infoE:
_ZTVN10__cxxabiv117__array_type_infoE:
_ZTVN10__cxxabiv117__class_type_infoE:
_ZTVN10__cxxabiv117__pbase_type_infoE:
_ZTVN10__cxxabiv119__pointer_type_infoE:
_ZTVN10__cxxabiv120__function_type_infoE:
_ZTVN10__cxxabiv120__si_class_type_infoE:
_ZTVN10__cxxabiv121__vmi_class_type_infoE:
_ZTVN10__cxxabiv123__fundamental_type_infoE:
_ZTVN10__cxxabiv129__pointer_to_member_type_infoE:
    .word 0
    .word 0
    .word 0
```

22.11 Further reading

Additional information on developing code for the Arm family of processors is available from both Arm and third parties.

Arm publications

Arm periodically provides updates and corrections to its documentation on *Arm Developer <https://developer.arm.com/>*. For example, current errata sheets and addenda, Arm Frequently Asked Questions (FAQs), and KnowledgeBase Articles (KBAs).

For full information about the base standard, software interfaces, and standards supported by Arm, see https://developer.arm.com/architectures/system-architectures/software-standards/abi.

In addition, see the following documentation for specific information relating to Arm® products:

- Arm Architecture Reference Manuals.
- Cortex-A series processors.
- Cortex-R series processors.
- Cortex-M series processors.
- Cortex-X series processors.
- Neoverse processors.

Other publications

This Arm Compiler for Embedded FuSa tools documentation is not intended to be an introduction to the C or C++ programming languages. It does not try to teach programming in C or C++, and it is not a reference manual for the C or C++ standards. Other publications provide general information about programming.

The following publications describe the C++ language:

- *ISO/IEC 14882:2017, C++ Standard*.
- Stroustrup, B., *The C++ Programming Language* (4th edition, 2013). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 978-0321563842.

The following publications provide general C++ programming information:

- Stroustrup, B., *The Design and Evolution of C++* (1994). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-54330-3.

  This book explains how C++ evolved from its first design to the language in use today.
- Vandevoorde, D and Josuttis, N.M. *C++ Templates: The Complete Guide* (2003). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-73484-2.
- Meyers, S., *Effective C++* (3rd edition, 2005). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 978-0321334879.

  This provides short, specific guidelines for effective C++ development.
- Meyers, S., *More Effective C++* (2nd edition, 1997). Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-92488-9.

The following publications provide general C programming information:

- ISO/IEC 9899:2011, *C Standard*.

  The standard is available from national standards bodies (for example, AFNOR in France, ANSI in the USA).
- Kernighan, B.W. and Ritchie, D.M., *The C Programming Language* (2nd edition, 1988). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-110362-8.

  This book is co-authored by the original designer and implementer of the C language, and is updated to cover the essentials of ANSI C.
- Harbison, S.P. and Steele, G.L., *A C Reference Manual* (5th edition, 2002). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-089592-X.

  This is a very thorough reference guide to C, including useful information on ANSI C.
- Plauger, P., *The Standard C Library* (1991). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-131509-9.

  This is a comprehensive treatment of ANSI and ISO standards for the C Library.
- Koenig, A., *C Traps and Pitfalls*, Addison-Wesley (1989), Reading, Mass. ISBN 0-201-17928-8.

  This explains how to avoid the most common traps in C programming. It provides informative reading at all levels of competence in C.

See https://www.dwarfstd.org for the latest information about the Debug With Arbitrary Record Format (DWARF) debug table standards and ELF specifications.

Appendix A Arm Compiler for Embedded FuSa User Guide Changes

Describes the technical changes that have been made to the Arm® Compiler for Embedded FuSa User Guide.

A.1 Changes for the Arm Compiler for Embedded FuSa User Guide

Changes that have been made to the Arm® Compiler for Embedded FuSa User Guide are listed with the latest version first.

Table A-1: Changes between 6.22.1 LTS and 6.22.2 LTS

Change	Topics affected
Added descriptions for the qualification status of GNU extensions.	• Extensions that are considered qualified features within the scope of functional safety certification • Extensions that are outside the scope of functional safety certification
Updated *C++ functions you can re-implement* with links to `-fthreadsafe-statics` and `-fno-threadsafe-statics`.	• C++ functions you can re-implement

Table A-2: Changes between 6.22 and 6.22.1 LTS

Change	Topics affected
Removed information about FlexNet licensing and replaced with user-based licensing information.	• System requirements and installation.
Added notes about incompatible mitigations with execute only.	• Security features supported in Arm Compiler for Embedded FuSa.
Moved the topic *Compiling with -mexecute-only generates an empty .text section* from the *Migration and Compatibility Guide*.	• Compiling with -mexecute-only generates an empty .text section.
Removed the environment variables that relied on FlexNet support, `ARM_PRODUCT_DEF`, `ARM_PRODUCT_PATH`, `ARM_TOOL_VARIANT`, and `ARMLMD_LICENSE_FILE`.	• Toolchain environment variables.
Added a section on using the `checksums.txt` file to verify the installation.	• System requirements and installation.
Added a section on installing Arm Compiler for Embedded FuSa as a standalone product on x86_64 Windows platforms and accepting the EULA.	• System requirements and installation.