Arm® Compiler

Version 6.15

User Guide

Arm® Compiler

User Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0606-00	04 November 2016	Non-Confidential	Arm Compiler v6.6 Release
0607-00	05 April 2017	Non-Confidential	Arm Compiler v6.7 Release
0608-00	30 July 2017	Non-Confidential	Arm Compiler v6.8 Release.
0609-00	25 October 2017	Non-Confidential	Arm Compiler v6.9 Release.
0610-00	14 March 2018	Non-Confidential	Arm Compiler v6.10 Release.
0611-00	25 October 2018	Non-Confidential	Arm Compiler v6.11 Release.
0612-00	27 February 2019	Non-Confidential	Arm Compiler v6.12 Release.
0613-00	09 October 2019	Non-Confidential	Arm Compiler v6.13 Release.
0614-00	26 February 2020	Non-Confidential	Arm Compiler v6.14 Release.
0615-00	07 October 2020	Non-Confidential	Arm Compiler v6.15 Release.
0615-01	14 December 2020	Non-Confidential	Documentation update 1 for Arm Compiler v6.15 Release.

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.

Web Address

developer.arm.com

Progressive terminology commitment

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

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact *terms@arm.com*.

Contents

Arm® Compiler User Guide

Preface

About this book 12

Chapter 1 Getting Started

1.1 Introduction to Arm® Compiler 6 1-16
1.2 About the Arm® Compiler toolchain assemblers 1-19
1.3 Installing Arm® Compiler 1-20
1.4 Accessing Arm® Compiler from Arm® Development Studio 1-21
1.5 Accessing Arm® Compiler from the Arm® Keil® µVision® IDE 1-22
1.6 Compiling a Hello World example 1-23
1.7 Using the integrated assembler 1-25
1.8 Running bare-metal images 1-27
1.9 Architectures supported by Arm® Compiler 1-28
1.10 Using Arm® Compiler securely in a shared environment 1-29

Chapter 2 Getting Started with the SVE features in Arm® Compiler

2.1 Introducing SVE 2-31
2.2 Assembling SVE code 2-32
2.3 Disassembling SVE object files 2-34
2.4 Running a binary in an AEMv8-A Base Fixed Virtual Platform (FVP) 2-35

Chapter 3 Using Common Compiler Options

3.1 Mandatory armclang options 3-39
3.2 Common Arm® Compiler toolchain options 3-41

3.3 Selecting source language options 3-44
3.4 Selecting optimization options 3-48
3.5 Building to aid debugging 3-52
3.6 Linking object files to produce an executable 3-53
3.7 Linker options for mapping code and data to target memory 3-54
3.8 Passing options from the compiler to the linker 3-55
3.9 Controlling diagnostic messages 3-56
3.10 Selecting floating-point options 3-61
3.11 Compilation tools command-line option rules 3-64

Chapter 4 Writing Optimized Code
4.1 Effect of the volatile keyword on compiler optimization 4-66
4.2 Optimizing loops 4-69
4.3 Inlining functions 4-74
4.4 Stack use in C and C++ 4-76
4.5 Packing data structures 4-79
4.6 Optimizing for code size or performance 4-83
4.7 Methods of minimizing function parameter passing overhead 4-85
4.8 Optimizing across modules with Link-Time Optimization 4-86
4.9 Scatter file section or object placement with Link-Time Optimization 4-91
4.10 How optimization affects the debug experience 4-96

Chapter 5 Assembling Assembly Code
5.1 Assembling armasm and GNU syntax assembly code 5-98
5.2 Preprocessing assembly code 5-100

Chapter 6 Using Assembly and Intrinsics in C or C++ Code
6.1 Using intrinsics 6-103
6.2 Writing inline assembly code 6-105
6.3 Calling assembly functions from C and C++ 6-108

Chapter 7 SVE Coding Considerations with Arm® Compiler
7.1 Embedding SVE assembly code directly into C and C++ code 7-111
7.2 Using SVE and SVE2 intrinsics directly in your C code 7-116

Chapter 8 Mapping Code and Data to the Target
8.1 What the linker does to create an image 8-123
8.2 Placing data items for target peripherals with a scatter file 8-125
8.3 Placing the stack and heap with a scatter file 8-126
8.4 Root region 8-127
8.5 Placing functions and data in a named section 8-130
8.6 Placing functions and data at specific addresses 8-132
8.7 Bare-metal Position Independent Executables 8-140
8.8 Placement of Arm® C and C++ library code 8-142
8.9 Placement of unassigned sections 8-144
8.10 Placing veneers with a scatter file 8-154
8.11 Preprocessing a scatter file 8-155
8.12 Reserving an empty block of memory 8-157
8.13 Aligning regions to page boundaries 8-159
8.14 Aligning execution regions and input sections 8-160

Chapter 9 **Overlays**

9.1 Overlay support in Arm® Compiler 9-162
9.2 Automatic overlay support 9-163
9.3 Manual overlay support 9-168

Chapter 10 **Embedded Software Development**

10.1 About embedded software development 10-177
10.2 Default compilation tool behavior 10-178
10.3 C library structure 10-179
10.4 Default memory map 10-180
10.5 Application startup 10-182
10.6 Tailoring the C library to your target hardware 10-183
10.7 Reimplementing C library functions 10-185
10.8 Tailoring the image memory map to your target hardware 10-187
10.9 About the scatter-loading description syntax 10-188
10.10 Root regions 10-189
10.11 Placing the stack and heap 10-190
10.12 Run-time memory models 10-191
10.13 Reset and initialization 10-193
10.14 The vector table 10-194
10.15 ROM and RAM remapping 10-195
10.16 Local memory setup considerations 10-196
10.17 Stack pointer initialization 10-197
10.18 Hardware initialization 10-198
10.19 Execution mode considerations 10-199
10.20 Target hardware and the memory map 10-200
10.21 Execute-only memory 10-201
10.22 Building applications for execute-only memory 10-202
10.23 Vector table for ARMv6 and earlier, ARMv7-A and ARMv7-R profiles 10-203
10.24 Vector table for M-profile architectures 10-204
10.25 Vector Table Offset Register 10-205
10.26 Integer division-by-zero errors in C code 10-206

Chapter 11 **Building Secure and Non-secure Images Using Arm®v8-M Security Extensions**

11.1 Overview of building Secure and Non-secure images 11-209
11.2 Building a Secure image using the Arm®v8-M Security Extensions 11-212
11.3 Building a Non-secure image that can call a Secure image 11-216
11.4 Building a Secure image using a previously generated import library 11-218

Chapter 12 **Overview of the Linker**

12.1 About the linker 12-223
12.2 armlink command-line syntax 12-226
12.3 What the linker does when constructing an executable image 12-227

Chapter 13 **Getting Image Details**

13.1 Options for getting information about linker-generated files 13-229
13.2 Identifying the source of some link errors 13-230
13.3 Example of using the --info linker option 13-231
13.4 How to find where a symbol is placed when linking 13-234

Chapter 14 SysV Dynamic Linking

14.1 Build a SysV shared object 14-236
14.2 Build a SysV executable 14-237

Chapter 15 Overview of the fromelf Image Converter

15.1 About the fromelf image converter 15-239
15.2 fromelf execution modes 15-240
15.3 Getting help on the fromelf command 15-241
15.4 fromelf command-line syntax 15-242

Chapter 16 Using fromelf

16.1 General considerations when using fromelf 16-244
16.2 Examples of processing ELF files in an archive 16-245
16.3 Options to protect code in image files with fromelf 16-246
16.4 Options to protect code in object files with fromelf 16-247
16.5 Option to print specific details of ELF files 16-249
16.6 Using fromelf to find where a symbol is placed in an executable ELF image 16-250

Chapter 17 Overview of the Arm® Librarian

17.1 About the Arm® Librarian 17-253
17.2 Considerations when working with library files 17-254
17.3 armar command-line syntax 17-255
17.4 Option to get help on the armar command 17-256

Chapter 18 Overview of the armasm Legacy Assembler

18.1 Key features of the armasm assembler 18-258
18.2 How the assembler works 18-259

Appendix A Supporting reference information

A.1 Support level definitions Appx-A-262
A.2 Standards compliance in Arm® Compiler Appx-A-266
A.3 Compliance with the ABI for the Arm® Architecture (Base Standard) Appx-A-267
A.4 GCC compatibility provided by Arm® Compiler 6 .... Appx-A-269
A.5 Locale support in Arm® Compiler Appx-A-270
A.6 Toolchain environment variables Appx-A-271
A.7 Clang and LLVM documentation Appx-A-273
A.8 Further reading Appx-A-274

Appendix B Arm® Compiler User Guide Changes

B.1 Changes for the Arm Compiler User Guide Appx-B-277

List of Figures
Arm® Compiler User Guide

Figure 1-1	A typical tool usage flow diagram	1-17
Figure 4-1	Structure without packing attribute or pragma	4-80
Figure 4-2	Structure with attribute packed	4-80
Figure 4-3	Structure with pragma pack with 1 byte alignment	4-80
Figure 4-4	Structure with pragma pack with 2 byte alignment	4-81
Figure 4-5	Structure with pragma pack with 4 byte alignment	4-81
Figure 4-6	Structure with attribute packed on individual member	4-81
Figure 4-7	Link-Time Optimization	4-86
Figure 8-1	Memory map for fixed execution regions	8-128
Figure 8-2	.ANY contingency	8-151
Figure 8-3	Reserving a region for the stack	8-158
Figure 10-1	C library structure	10-179
Figure 10-2	Default memory map	10-180
Figure 10-3	Linker placement rules	10-180
Figure 10-4	Default initialization sequence	10-182
Figure 10-5	Retargeting the C library	10-183
Figure 10-6	Scatter-loading description syntax	10-188
Figure 10-7	One-region model	10-191
Figure 10-8	Two-region model	10-192
Figure 10-9	Initialization sequence	10-193
Figure A-1	Integration boundaries in Arm Compiler 6.	Appx-A-264

List of Tables
Arm® Compiler User Guide

Table 3-1	armclang common options	3-41
Table 3-2	armlink common options	3-42
Table 3-3	armar common options	3-43
Table 3-4	fromelf common options	3-43
Table 3-5	armasm common options	3-43
Table 3-6	Source language variants	3-44
Table 3-7	Exceptions to the support for the language standards	3-45
Table 3-8	Optimization goals	3-48
Table 3-9	Example code generation with -O0	3-50
Table 3-10	Example code generation with -O1	3-51
Table 3-11	armclang linker control options	3-55
Table 3-12	Common diagnostic options	3-56
Table 3-13	Options for floating-point selection	3-61
Table 3-14	Floating-point linkage for AArch32	3-63
Table 4-1	C code for nonvolatile and volatile buffer loops	4-67
Table 4-2	Disassembly for nonvolatile and volatile buffer loop	4-67
Table 4-3	Loop unrolling pragmas	4-69
Table 4-4	Loop optimizing example	4-69
Table 4-5	Loop examples	4-70
Table 4-6	Example loops	4-70
Table 4-7	Assembly code from vectorizable and non-vectorizable loops	4-71
Table 4-8	C code for incrementing and decrementing loops	4-72
Table 4-9	C disassembly for incrementing and decrementing loops	4-72

Table 4-10 Function inlining 4-74
Table 4-11 Effect of -fno-inline-functions 4-75
Table 4-12 Packing members in a structure or union 4-79
Table 4-13 Packing structures 4-80
Table 4-14 Packing individual members 4-81
Table 7-1 Element selection by predicate type svbool_t 7-116
Table 7-2 Common addressing mode disambiguators 7-118
Table 8-1 Input section properties for placement of .ANY sections 8-147
Table 8-2 Input section properties for placement of sections with next_fit 8-148
Table 8-3 Input section properties and ordering for sections_a.o and sections_b.o 8-149
Table 8-4 Sort order for descending_size algorithm 8-150
Table 8-5 Sort order for cmdline algorithm 8-150
Table 9-1 Using relative offset in overlays 9-169
Table A-1 Environment variables used by the toolchain Appx-A-271
Table B-1 Changes between 6.15 and 6.14 .... Appx-B-277

Preface

This preface introduces the *Arm® Compiler User Guide*.

It contains the following:
- *About this book* on page 12.

About this book

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

Using this book

This book is organized into the following chapters:

Chapter 1 Getting Started
This chapter introduces Arm Compiler 6 and helps you to start working with Arm Compiler 6 quickly. You can use Arm Compiler 6 from Arm Development Studio, Arm DS-5 Development Studio, Arm Keil® MDK, or as a standalone product.

Chapter 2 Getting Started with the SVE features in Arm® Compiler
Describes how to generate an executable binary that makes use of the instructions provided by the SVE architectural extension to the Armv8-A architecture.

Chapter 3 Using Common Compiler Options
There are many options that you can use to control how Arm Compiler generates code for your application. This section lists the mandatory and commonly used optional command-line arguments, such as to control target selection, optimization, and debug view.

Chapter 4 Writing Optimized Code
To make best use of the optimization capabilities of Arm Compiler, there are various options, pragmas, attributes, and coding techniques that you can use.

Chapter 5 Assembling Assembly Code
Describes how to assemble assembly source code with `armclang` and `armasm`.

Chapter 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 which C or C++ provides.

Chapter 7 SVE Coding Considerations with Arm® Compiler
Describes best practices for writing code that uses the SVE and SVE2 features of Arm Compiler.

Chapter 8 Mapping Code and Data to the Target
There are various options in Arm Compiler to control how code, data and other sections of the image are mapped to specific locations on the target.

Chapter 9 Overlays
Describes the Arm Compiler support for overlays to enable you to have multiple load regions at the same address.

Chapter 10 Embedded Software Development
Describes how to develop embedded applications with Arm Compiler, with or without a target system present.

Chapter 11 Building Secure and Non-secure Images Using Arm®v8-M Security Extensions
Describes how to use the Armv8-M Security Extensions to build a secure image, and how to allow a non-secure image to call a secure image.

Chapter 12 Overview of the Linker
Gives an overview of the Arm linker, `armlink`.

Chapter 13 Getting Image Details
Describes how to get image details from the Arm linker, `armlink`.

Chapter 14 SysV Dynamic Linking
Arm Compiler 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.

Chapter 15 Overview of the fromelf Image Converter
Gives an overview of the `fromelf` image converter provided with Arm Compiler.

Chapter 16 Using fromelf
Describes how to use the `fromelf` image converter provided with Arm Compiler.

Chapter 17 Overview of the Arm® Librarian
Gives an overview of the Arm Librarian, `armar`, provided with Arm Compiler.

Chapter 18 Overview of the armasm Legacy Assembler
Gives an overview of the `armasm` legacy assembler provided with Arm Compiler toolchain.

Appendix A Supporting reference information
The various features in Arm Compiler might have different levels of support, ranging from fully supported product features to community features.

Appendix B Arm® Compiler User Guide Changes
Describes the technical changes that have been made to the Arm Compiler User Guide.

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.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
Denotes language keywords when used outside example code.

`<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
Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Arm Compiler User Guide*.
- The number 100748_0615_01_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

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

Chapter 1
Getting Started

This chapter introduces Arm Compiler 6 and helps you to start working with Arm Compiler 6 quickly. You can use Arm Compiler 6 from Arm Development Studio, Arm DS-5 Development Studio, Arm Keil® MDK, or as a standalone product.

It contains the following sections:

- *1.1 Introduction to Arm® Compiler 6* on page 1-16.
- *1.2 About the Arm® Compiler toolchain assemblers* on page 1-19.
- *1.3 Installing Arm® Compiler* on page 1-20.
- *1.4 Accessing Arm® Compiler from Arm® Development Studio* on page 1-21.
- *1.5 Accessing Arm® Compiler from the Arm® Keil® µVision® IDE* on page 1-22.
- *1.6 Compiling a Hello World example* on page 1-23.
- *1.7 Using the integrated assembler* on page 1-25.
- *1.8 Running bare-metal images* on page 1-27.
- *1.9 Architectures supported by Arm® Compiler* on page 1-28.
- *1.10 Using Arm® Compiler securely in a shared environment* on page 1-29.

1.1 Introduction to Arm® Compiler 6

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

Arm Compiler 6 is a component of *Arm® Development Studio*, *Arm® DS-5 Development Studio*, and *Arm® Keil® MDK*. Alternatively, you can use Arm Compiler 6 as a *standalone product*. The features and processors that Arm Compiler 6 supports depend on the product edition. See *Compare Editions* for Arm Development Studio and *Arm® DS-5 Development Studio editions* for the specification of the different standard products.

Arm Compiler 6 combines the optimized tools and libraries from Arm with a modern LLVM-based compiler framework. The components in Arm Compiler 6 are:

armclang

The compiler and integrated assembler that compiles C, C++, and GNU 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` assembler and GNU syntax for all new assembly files.

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.

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.

Note

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

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.

Application development

A typical application development flow might involve the following:

- 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.

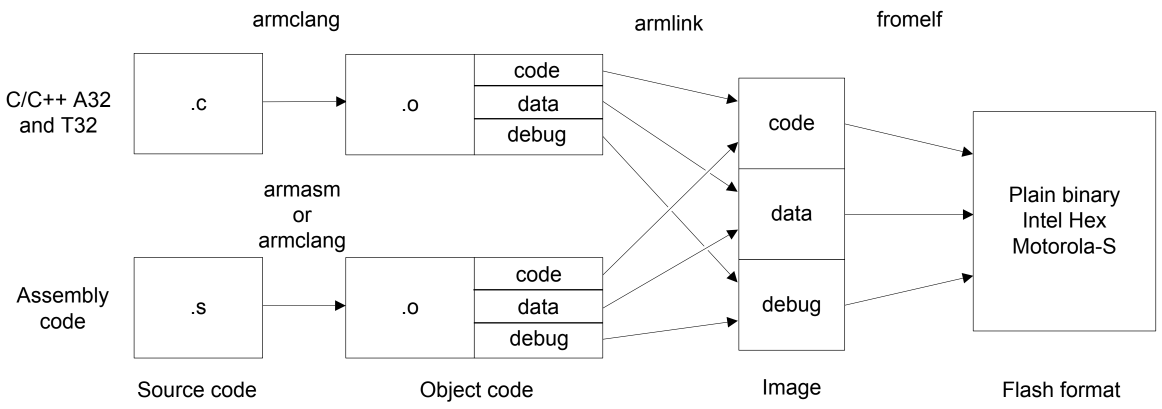


Figure 1-1 A typical tool usage flow diagram

Arm Compiler 6 has more functionality than the set of product features that is described in the documentation. The various features in Arm Compiler 6 can have different levels of support and guarantees. For more information, see *Support level definitions* on page Appx-A-262.

Note

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

Note

Be aware of the following:

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

Related concepts

1.6 Compiling a Hello World example on page 1-23

Related references

3.2 Common Arm® Compiler toolchain options on page 3-41

Related information

-S (armclang)

1.2 About the Arm® Compiler toolchain assemblers

The Arm Compiler 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 command-line option descriptions and related information in the *Arm® Compiler Reference Guide* describe all the features that Arm Compiler 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* on page Appx-A-262 is operating correctly.

Related concepts

5.1 Assembling armasm and GNU syntax assembly code on page 5-98

Related references

Chapter 6 Using Assembly and Intrinsics in C or C++ Code on page 6-101

Related information

Arm Compiler Reference Guide

1.3 Installing Arm® Compiler

This topic lists the system requirements for running Arm Compiler and guides you through the installation process.

System Requirements

Arm Compiler 6 is available for the following operating systems:

- Windows 64-bit.
- Windows 32-bit.
- Linux 64-bit.

For more information on system requirements see the *Arm® Compiler release note*.

Installing Arm® Compiler

You can install Arm Compiler as a standalone product on supported Windows and Linux platforms. If you use Arm Compiler as part of a development suite such as Arm Development Studio, Arm DS-5 Development Studio, or Arm Keil MDK, installing the development suite also installs Arm Compiler. The following instructions are for installing Arm Compiler as a standalone product.

Prerequisites:

1. *Download Arm® Compiler 6*.
2. Obtain a license. Contact your Arm sales representative or *request a license*.

Installing a standalone Arm® Compiler on Windows platforms

To install Arm Compiler as a standalone product on Windows, you need the `setup.exe` installer on your machine. This is in the *Arm® Compiler 6 download*:

1. On 64-bit platforms, run `win-x86_64\setup.exe`. On 32-bit platforms, run `win-x86_32\setup.exe`.
2. Follow the on-screen installation instructions.
3. Some license types require you to complete further configuration steps. To check if your license requires further configuration, and to learn how to configure that license, see *Arm® Compiler Licensing Configuration*.

If you have an older version of Arm Compiler 6 and you want to upgrade, Arm recommends that you uninstall the older version of Arm Compiler 6 before installing the new version of Arm Compiler 6.

Installing a standalone Arm® Compiler on Linux platforms

To install Arm Compiler as a standalone product on Linux platforms, you need the `install_x86_64.sh` installer on your machine. This is in the *Arm® Compiler 6 download*:

1. Run `install_x86_64.sh` normally, without using the `source` Linux command.
2. Follow the on-screen installation instructions.
3. Some license types require you to complete further configuration steps. To check if your license requires further configuration, and to learn how to configure that license, see *Arm® Compiler Licensing Configuration*.

Uninstalling a standalone Arm® Compiler

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

1. Select **Control Panel** > **Programs and Features**.
2. Select the version that you want to uninstall, for example **Arm Compiler 6.10**.
3. Click the **Uninstall** button.

To uninstall Arm Compiler 6 installation directory for the compiler version you want to delete.

For more information on installation, see the *Arm® Compiler release note*.

Related concepts

1.4 Accessing Arm® Compiler from Arm® Development Studio on page 1-21
1.5 Accessing Arm® Compiler from the Arm® Keil® µVision® IDE on page 1-22

1.4 Accessing Arm® Compiler from Arm® Development Studio

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

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

Related references

1.3 Installing Arm® Compiler on page 1-20

1.5 Accessing Arm® Compiler from the Arm® Keil® µVision® IDE

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

For more information, see *Manage Arm® Compiler Versions* in the *µVision® User's Guide*.

Related references

1.3 Installing Arm® Compiler on page 1-20

1.6 Compiling a Hello World example

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

The source code

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;
}
```

Compiling in a single step

When compiling code, 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, `a.out`.

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 Reference Guide* describes all the supported options. Some of the most common options are listed in *3.2 Common Arm® Compiler toolchain options* on page 3-41.

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 and linking in a single step might be the simplest option:

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

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

More complex projects might have many more source files. It is not efficient to compile every source file at every compilation, because most 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 the 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

1.7 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 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 *Arm® Application Procedure Call Standard* (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 `--target` option determines which target state to compile for. This option is a mandatory option.

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

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

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

The `--target` option selects the target that you want to assemble for. 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.

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 code is marked as T32 by default 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 concepts

3.1 Mandatory armclang options on page 3-39

Related information

Summary of armclang command-line options

Sections

1.8 Running bare-metal images

By default, Arm Compiler 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.

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 User Guide*.
- For Arm DS-5, see the *Arm® DS-5 Debugger User Guide*.

By default, the C library in Arm Compiler 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 User Guide*.
- For Arm DS-5 Debugger settings, see *Configuring a connection to a bare-metal hardware target* in the *Arm® DS-5 Debugger User Guide*.

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 or Arm DS-5 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 User Guide
Arm DS-5 Debugger User Guide
Semihosting for AArch32 and AArch64

1.9 Architectures supported by Arm® Compiler

Arm Compiler supports a number of different architecture profiles.

Arm Compiler supports the following architectures:

- Armv8-A and all update releases, for bare-metal targets.
- Armv8-R.
- Armv8-M.
- Armv7-A for bare-metal targets.
- Armv7-R.
- Armv7-M.
- Armv6-M.

When compiling code, the compiler needs to know which architecture to target in order 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 `armasm` and `armlink`, use the `--cpu` option to specify target processors and architectures.

Related information

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

1.10 Using Arm® Compiler securely in a shared environment

Arm Compiler 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 into environments where security is a concern, then Arm strongly recommends that you do all of 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.

Chapter 2
Getting Started with the SVE features in Arm® Compiler

Describes how to generate an executable binary that makes use of the instructions provided by the SVE architectural extension to the Armv8-A architecture.

It contains the following sections:

- *2.1 Introducing SVE* on page 2-31.
- *2.2 Assembling SVE code* on page 2-32.
- *2.3 Disassembling SVE object files* on page 2-34.
- *2.4 Running a binary in an AEMv8-A Base Fixed Virtual Platform (FVP)* on page 2-35.

2.1 Introducing SVE

The Arm Compiler 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 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 toolchain only supports bare-metal applications. For SVE compilation for Linux, use Arm Compiler for Linux, that is part of *Arm Allinea Studio*.

——— Note ———

Arm Compiler does not support auto-vectorization for SVE. If you require auto-vectorization for SVE, then you must use Arm Compiler for Linux. For more information, see *Arm Allinea Studio*.

Related information

Arm Compiler 6 documentation

2.2 Assembling SVE code

Use `armclang` with a suitable SVE-enabled target to assemble code containing SVE instructions.

The SVE architectural extension to the Armv8-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 tasks

2.3 Disassembling SVE object files on page 2-34

Related information

Arm Compiler Reference Guide

armclang -c option

armclang -o option

armclang -march option
armclang --target option

2.3 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.

To disassemble an ELF-format object file containing SVE instructions, use `fromelf` with the `-c` option.

Procedure

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

```
#ifdef __ARM_FEATURE_SVE
#include <arm_sve.h>
#endif /* __ARM_FEATURE_SVE */

void daxpy_1_1(int64_t n, double da, double *dx, double *dy)
{
    int64_t i = 0;
    svbool_t pg = svwhilelt_b64(i, n);
    do {
        svfloat64_t dx_vec = svld1(pg, &dx[i]);
        svfloat64_t dy_vec = svld1(pg, &dy[i]);
        svst1(pg, &dy[i], svmla_x(pg, dy_vec, dx_vec, da));
        i += svcntd();
        pg = svwhilelt_b64(i, n);
    }
    while (svptest_any(svptrue_b64(), pg));
}
```

2. Compile and use `fromelf` to view the disassembly:

```
armclang -c -O3 --target=aarch64-arm-none-eabi -march=armv8-a+sve -o daxpy.o daxpy.c
fromelf -c daxpy.o
```

Results:

The disassembly is as follows:

```
...
** Section #3 '.text.daxpy_1_1' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 76 bytes (alignment 4)
    Address: 0x00000000

    $x.0
    daxpy_1_1
        0x00000000:    04e0e3e9    ....    CNTD     x9
        0x00000004:    aa1f03e8    ....    MOV      x8,xzr
        0x00000008:    25e017e0    ...%    WHILELT  p0.D,xzr,x0
        0x0000000c:    05282000    . (.    MOV      z0.D,d0
        0x00000010:    25d8e3e1    ...%    PTRUE    p1.D
        0x00000014:    04e7e3ea    ....    CNTD     x10,ALL,MUL #8
        0x00000018:    aa0903eb    ....    MOV      x11,x9
        0x0000001c:    8b08002c    ,...    ADD      x12,x1,x8
        0x00000020:    8b08004d    M...    ADD      x13,x2,x8
        0x00000024:    a5e0a181    ....    LD1D     {z1.D},p0/Z,[x12]
        0x00000028:    a5e0a1a2    ....    LD1D     {z2.D},p0/Z,[x13]
        0x0000002c:    8b0a0108    ....    ADD      x8,x8,x10
        0x00000030:    65e00022    "..e    FMLA     z2.D,p0/M,z1.D,z0.D
        0x00000034:    e5e0e1a2    ....    ST1D     {z2.D},p0,[x13]
        0x00000038:    25e01560    `..%    WHILELT  p0.D,x11,x0
        0x0000003c:    2550c400    ..P%    PTEST    p1,p0.B
        0x00000040:    8b09016b    k...    ADD      x11,x11,x9
        0x00000044:    54fffec1    ...T    B.NE     {pc}-0x28 ; 0x1c
        0x00000048:    d65f03c0    .._.    RET
...
```

Related concepts

2.2 Assembling SVE code on page 2-32

2.4 Running a binary in an AEMv8-A Base Fixed Virtual Platform (FVP)

Describes how to compile a program with Arm Compiler and then run the resulting binary using the AEMv8-A 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. The `italic` elements indicate parameters that can be edited.

```
$FVP_BASE/FVP_Base_AEMv8A-AEMv8A \
  --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:

```
Total instructions executed: 10344
User time:    0.01 sec
Kernel time: 0.00 sec
CPU time:    0.01 sec
Elapsed clock: 0.00 sec
```

`$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 uses the `svld1`, `svst1`, `svcntd`, `svmla_x`, and `svwhilelt_b64` intrinsics:

```
// daxpy_acle.c
#include <stdio.h>
#ifdef __ARM_FEATURE_SVE
#include <arm_sve.h>
#endif /* __ARM_FEATURE_SVE */

void daxpy_1_1(int64_t n, double da, double *dx, double *dy)
{
    int64_t i = 0;
    svbool_t pg = svwhilelt_b64(i, n);
    do {
        svfloat64_t dx_vec = svld1(pg, &dx[i]);
        svfloat64_t dy_vec = svld1(pg, &dy[i]);
        svst1(pg, &dy[i], svmla_x(pg, dy_vec, dx_vec, da));
        i += svcntd();
        pg = svwhilelt_b64(i, n);
    }
    while (svptest_any(svptrue_b64(), pg));
}

int main(int argc, char* argv[]) {
  double da = 1.5;
  double *dx;
  double *dy;

  *dx = 1.5;
  *dy = 1.5;
  daxpy_1_1(10, da, dx, dy);
  return 0;
}
```

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 daxpy_acle.axf daxpy_acle.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 will be [veclen] x 64 bits. Only
#    even values of veclen are valid.
#
#
# Set the FVP_BASE environment variable to point to the FVP directory.
#
# Set the ARMLMD_LICENSE_FILE environment variable to reference a license
# file or license server with entitlement for the FVP.

VECLEN=$1
CMDLINE=$2

$FVP_BASE/FVP_Base_AEMv8A-AEMv8A \
   --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.

Related information

Arm Compiler Reference Guide

armclang -o option
armclang -Xlinker option
armclang -O option
armclang -march option
armclang --target option

Chapter 3
Using Common Compiler Options

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

It contains the following sections:

- *3.1 Mandatory armclang options* on page 3-39.
- *3.2 Common Arm® Compiler toolchain options* on page 3-41.
- *3.3 Selecting source language options* on page 3-44.
- *3.4 Selecting optimization options* on page 3-48.
- *3.5 Building to aid debugging* on page 3-52.
- *3.6 Linking object files to produce an executable* on page 3-53.
- *3.7 Linker options for mapping code and data to target memory* on page 3-54.
- *3.8 Passing options from the compiler to the linker* on page 3-55.
- *3.9 Controlling diagnostic messages* on page 3-56.
- *3.10 Selecting floating-point options* on page 3-61.
- *3.11 Compilation tools command-line option rules* on page 3-64.

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 Armv8-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 Reference Guide* for a *list of architecture features* that Arm Compiler 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 this 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. Arm recommends that you always specify a suitable optimization level. For more information, see *Selecting optimization options* in the *Arm® Compiler User Guide*, and see the *`-O`* option in the *Arm® Compiler 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 toolchain options

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

armclang common options

See the *Arm Compiler 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.
-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 is 32-bit instructions. 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. The compiler ignores the `-marm` option and generates a warning with these targets.
-mthumb	Requests that the compiler targets the T32 instruction set, which is mixed 32-bit and 16-bit 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 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	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.

Table 3-1 armclang common options (continued)

Option	Description
-o	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 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.
--info	Displays information about linker operation. For example, `--info=sizes,unused,unusedsymbols` displays information about all of 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 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`.
`-d 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 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.
`--info`	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 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`	Generates DWARF debug tables compatible with the DWARF 3 standard.
`--fpu=name`	Selects the target floating-point unit (FPU) architecture.
`-o`	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 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 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* on page Appx-A-262.

Source language

By default Arm Compiler 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 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 supports Standard and GNU variants of source languages as shown in the following table.

Table 3-6 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 [COMMUNITY]	gnu++17 [COMMUNITY]

The default language standard for C code is `gnu11` [COMMUNITY]. The default language standard for C++ code is `gnu++14`. To specify a different source language standard, use the `-std=`*`name`* option.

Arm Compiler supports various language extensions, including GCC extensions, which you can use in your source code. The GCC extensions are only available when you specify one of the GCC C or C++ language variants. For more information on language extensions, see the *Arm® C Language Extensions* in Arm Compiler.

Since Arm Compiler 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 does not support strict adherence to C++98 or C++03.

If you do not use `-Wpedantic`, Arm Compiler 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 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 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 6 with `libc++` provides varying levels of support for different source language standards. The following table lists the exceptions to the support Arm Compiler 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.

Table 3-7 Exceptions to the support for the language standards (continued)

Language standard	Exceptions to the support for the language standard
C++98	• The C++98 standard is supported except where: — C++11 deviates from C++98. For example, where `std::deque<T>::insert()` returns an iterator, as required by the C++11 standard, but the C++98 standard requires it to return `void`. Information about how the C++11 standard deviates from the C++98 standard is available in Annex "C Compatibility" of the C++11 standard definition. — Where the `libc++` library deviates from the C++98 standard library: ◦ For `std::vector<T>::const_reference`, the C++98 standard requires the `const_reference` type to be `bool`, but in `libc++` it is an implementation-defined, read-only bit reference class. ◦ For `std::bitset<N>`, the C++98 standard requires `bool operator[] (size_t pos) const;` to return `bool`, but in `libc++` it returns an implementation-defined, read-only bit reference object. ◦ For `std::raw_storage_iterator`, the C++98 standard requires the `raw_storage_iterator` class template to be inherited from `std::iterator<std::output_iterator_tag,void,void,void,void>`, but in `libc++` it is inherited from an instantiation of `std::iterator` with a different list of template arguments. • Support for `-fno-exceptions` is limited.
C++03	• The C++03 standard is supported except where: — C++11 deviates from C++03. 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::vector<T>::const_reference`, the C++03 standard requires the `const_reference` type to be `bool`, but in `libc++` it is an implementation-defined, read-only bit reference class. ◦ For `std::bitset<N>`, the C++03 standard requires `bool operator[] (size_t pos) const;` to return `bool`, but in `libc++` it returns an implementation-defined, read-only bit reference object. ◦ 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>`, but in `libc++` it is inherited from an instantiation of `std::iterator` with a different list of template arguments. • Support for `-fno-exceptions` is limited.
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 `thread_local` keyword is not supported.

Table 3-7 Exceptions to the support for the language standards (continued)

Language standard	Exceptions to the support for the language standard
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 `thread_local` keyword is not supported. **Note** gnu++14 is the default language standard for C++ code.
C++17 [COMMUNITY]	The base Clang and `libc++` components provide C++17 language functionality. However, Arm has performed no independent testing of these features and therefore these features are [COMMUNITY] features.

Additional information

See the *Arm Compiler 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*.

The Clang documentation provides additional information about language compatibility:
- Language compatibility:

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

  http://clang.llvm.org/docs/LanguageExtensions.html
- C++ status:

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

Arm® Compiler 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 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

Standard C++ library implementation definition

Arm Compiler Reference Guide

3.4 Selecting optimization options

Arm Compiler performs several optimizations to reduce the code size and improve the performance of your application. There are different optimization levels which 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 provides various optimization levels to control the different optimization goals. The best optimization level for your application depends on your application and optimization goal.

Table 3-8 Optimization goals

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 which matters most for your application, and then choose an option which 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`. Arm recommends this option for a good debug experience.

The differences when using `-O1`, as compared to `-O0` are:
- Optimizations are enabled, which might reduce the fidelity of debug information.
- Inlining and tail calls are enabled, meaning backtraces might not give the stack of open function activations which might be expected 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 have been 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). Arm recommends 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:

- Instructs the compiler to optimize for code size only and disregard the 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.

Optimization level -Omin

`-Omin` aims to provide the smallest possible code size. Arm recommends this option for best code size. This optimization level degrades the debug experience compared to `-O1`.

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

- `-Omin` enables a basic set of Link-Time Optimizations (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.

Optimization level -Ofast

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

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 Link-Time Optimization (LTO).

At this optimization level, Arm Compiler 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.

Examples

The example shows the code generation when using the `-O0` optimization option. To perform this optimization, compile your source file using:

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

Table 3-9 Example code generation with -O0

Source code in file.c	Unoptimized output from `armclang`

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

The example shows the code generation when using the `-O1` optimization option. To perform this optimization, compile your source file using:

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

Table 3-10 Example code generation with -O1

Source code in file.c	Optimized output from `armclang`

`int dummy()`
`{`
`    int x=10, y=20;`
`    int z;`
`    z=x+y;`
`    return 0;`
`}`
	`dummy:`
	`    .fnstart`
	`    movs r0, #0`
	`    bx lr`

The source file contains mostly dead code, such as `int x=10` and `z=x+y`. At optimization level `-O0`, the compiler performs no optimization, and therefore generates code for the dead code in the source file. However, at optimization level `-O1`, the compiler does not generate code for the dead code in the source file.

Related information

Semihosting for AArch32 and AArch64

3.5 Building to aid debugging

During application development, you must debug the image that you build. The Arm Compiler 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 or DWARF 3 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. Arm recommends 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 optimization levels `-O2` and `-O3`. If you must use these optimization levels, 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 --no_debug` option.
- The `armlink --inline` option. 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
```

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 Armv7-M or Armv8-M architectures, or objects that are built with the `armclang -mthumb` option,

- 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 3-11 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 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 lists all the warnings and errors it encounters during the compiling and linking process. However, if you specify multiple source files, Arm Compiler only reports diagnostic information for the first source file that it encounters an error in.

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 3-12 Common diagnostic options

Option	Description
`-Werror`	Turn all warnings into errors.
`-Werror=foo`	Turn warning flag `foo` into an error.
`-Wno-error=foo`	Leave warning flag `foo` as a warning even if `-Werror` is specified.
`-Wfoo`	Enable warning flag `foo`.
`-Wno-foo`	Suppress warning flag `foo`.
`-w`	Suppress all warnings. Note that this option is a lowercase `w`.

Table 3-12 Common diagnostic options (continued)

Option	Description
`-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 *Controlling Errors and Warnings* in the *Clang Compiler User's Manual* for full details about controlling diagnostics with `armclang`.

Examples of controlling diagnostic messages with `armclang`

Copy the following code example to `file.c` and compile it with Arm Compiler 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. Therefore `armclang` generates this diagnostic message:

```
file.c:9:36: warning: more '%' conversions than data arguments [-Wformat]
                printf("Result of %d plus %d is %d\n", i, x);
                                                     ^
```

By default, `armclang` compiles for the `gnu11` standard for `.c` files. This language standard does not allow implicit function declarations. Therefore `armclang` generates this diagnostic message:

```
file.c:11:3: warning: implicit declaration of function 'call' is invalid C99 [-Wimplicit-
function-declaration]
    call();
    ^
```

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 Errors and Warnings* 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 "-Wname"`

Ignores the diagnostic message specified by *name*.

`#pragma clang diagnostic warning "-Wname"`

Sets the diagnostic message specified by *name* to warning severity.

`#pragma clang diagnostic error "-Wname"`

Sets the diagnostic message specified by *name* to error severity.

`#pragma clang diagnostic fatal "-Wname"`

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, `-Wname`, to suppress or change the severity of messages, but the change applies for the entire compilation.

Example of using pragmas to selectively override a command-line option

`foo.c:`

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

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

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

#pragma clang diagnostic pop
```

If you build this example with:

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

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

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

`--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 Errors and Warnings Reference Guide*.

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

```
$ armasm test.s --cpu=8-A.32
"test.s", line 55: Warning: A1313W: Missing END directive at end of file
0 Errors, 1 Warning

$ armasm test.s --cpu=8-A.32 --diag_remark=A1313
"test.s", line 55: 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 supports floating-point arithmetic and floating-point data types in your source code or application.

Arm Compiler 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 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 3-13 Options for floating-point selection

Option	Description
`armclang -mfpu`	Specify the floating-point architecture to the compiler (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 this 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 uses the available floating-point hardware that is based on the target you specify for `-mcpu` or `-march`. However, you can force Arm Compiler to disable the floating-point hardware. Disabling floating-point hardware forces Arm Compiler 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 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
  ```
- Software floating-point library for AArch64 is not currently available. Therefore, if you disable floating-point hardware when compiling for AArch64 targets, Arm Compiler does not support floating-point arithmetic in your source code.
- Disabling floating-point arithmetic does not disable all the floating-point hardware because the floating-point hardware is also used for Advanced 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* and *-mcpu* in the *Arm Compiler Reference Guide* for more information.

When compiling for AArch32:

- By default, Arm Compiler 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*, *-mcpu*, and *-mfpu* in the *Arm Compiler 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, Arm Compiler always uses hardware linkage. When using hardware linkage, Arm Compiler passes and returns floating-point values in hardware floating-point registers.

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

Table 3-14 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 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 Reference Guide* for more information on how the *--fpu* option specifies the linkage type and floating-point hardware.

Related information

-mcpu (armclang)

-mfloat-abi (armclang)

-mfpu (armclang)

About 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)'
```

Chapter 4
Writing Optimized Code

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

It contains the following sections:

- *4.1 Effect of the volatile keyword on compiler optimization* on page 4-66.
- *4.2 Optimizing loops* on page 4-69.
- *4.3 Inlining functions* on page 4-74.
- *4.4 Stack use in C and C++* on page 4-76.
- *4.5 Packing data structures* on page 4-79.
- *4.6 Optimizing for code size or performance* on page 4-83.
- *4.7 Methods of minimizing function parameter passing overhead* on page 4-85.
- *4.8 Optimizing across modules with Link-Time Optimization* on page 4-86.
- *4.9 Scatter file section or object placement with Link-Time Optimization* on page 4-91.
- *4.10 How optimization affects the debug experience* on page 4-96.

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 externally to the implementation, for example:

- By the operating system.
- By another thread of execution such as an interrupt routine or signal handler.
- 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 does not guarantee that a single-copy atomic instruction is used to access a **`volatile`** variable when one is available for the target processor, but not guaranteed to be single-copy atomic by the architecture.

———————————

When to use volatile

Use the **`volatile`** keyword for variables that might be modified from outside the scope that they are defined in.

For example, an external process might update a variable in a function. But if the variable appears unmodified, then the compiler might use the value of the older variable that is saved in a register rather than accessing it from memory. Declaring the variable as **`volatile`** makes the compiler access this variable from memory whenever the variable is referenced in code. This declaration ensures that the code always uses the updated variable value from memory.

Another example is that 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 practice, you must declare a variable as **`volatile`** when:

- Accessing memory-mapped peripherals.
- Sharing global variables between multiple threads.
- Accessing global variables in an interrupt routine or signal handler.

You should 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.
- Multi-threaded code might exhibit strange behavior.
- Optimization might result in the removal of code that implements deliberate timing delays.

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 of infinite loop when not using the volatile keyword

The use of the **`volatile`** keyword is illustrated in the two example routines in the following table.

Table 4-1 C code for nonvolatile and volatile buffer loops

Nonvolatile version of buffer loop	Volatile version of buffer loop

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

Both of these routines increment 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.

The example on the left does not declare the variable `buffer_full` as **`volatile`** and is therefore wrong. The example on the right does declare the variable `buffer_full` as **`volatile`**.

The following table shows the corresponding disassembly of the machine code that the compiler produces for each of the examples in *Table 4-1 C code for nonvolatile and volatile buffer loops* on page 4-67. The C code for each example is compiled using `armclang --target=arm-arm-none-eabi -march=armv8-a -Os -S`.

Table 4-2 Disassembly for nonvolatile and volatile buffer loop

Nonvolatile version of buffer loop	Volatile version of buffer loop

`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`
	`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 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`.

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

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 4-3 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. Arm recommends that you use `#pragma unroll` instead of manually unrolling loops. See *#pragma unroll[(n)], #pragma unroll_completely* in the *Arm® Compiler Reference Guide* for more information.

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

Table 4-4 Loop optimizing example

Bit counting loop without unrolling	Bit counting loop with unrolling

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

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

The following code is the code that Arm Compiler generates for the preceding examples. Copy the examples into `file.c` and compile using:

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

For the function with loop unrolling, `countSetBits2`, the generated code is faster but larger in size.

Table 4-5 Loop examples

Bit counting loop without unrolling

```
countSetBits1:
        mov     r1, r0
        mov     r0, #0
        cmp     r1, #0
        bxeq    lr
        mov     r2, #0
        mov     r0, #0
.LBB0_1:
        and     r3, r1, #1
        cmp     r2, r1, asr #1
        add     r0, r0, r3
        lsr     r3, r1, #1
        mov     r1, r3
        bne     .LBB0_1
        bx      lr
```

Bit counting loop with unrolling

```
countSetBits2:
        mov     r1, r0
        mov     r0, #0
        cmp     r1, #0
        bxeq    lr
        mov     r2, #0
        mov     r0, #0
LBB0_1:
        and     r3, r1, #1
        cmp     r2, r1, asr #1
        add     r0, r0, r3
        beq     .LBB0_4
@ BB#2:
        asr     r3, r1, #1
        cmp     r2, r1, asr #2
        and     r3, r3, #1
        add     r0, r0, r3
        asrne   r3, r1, #2
        andne   r3, r3, #1
        addne   r0, r0, r3
        cmpne   r2, r1, asr #3
        beq     .LBB0_4
@ BB#3:
        asr     r3, r1, #3
        cmp     r2, r1, asr #4
        and     r3, r3, #1
        add     r0, r0, r3
        asr     r3, r1, #4
        mov     r1, r3
        bne     .LBB0_1
.LBB0_4:
        bx    lr
```

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

Loop vectorization

If your target has the Advanced SIMD unit, then Arm Compiler 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* in the *Arm® Compiler 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.

For example, loops that access structures can be vectorized if all parts of the structure are accessed within the same loop rather than in separate loops. The following examples show a loop that Advanced SIMD can vectorize, and a loop that cannot be vectorized easily.

Table 4-6 Example loops

Vectorizable by Advanced SIMD

```
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;
  }
}
```

Not vectorizable by Advanced SIMD

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

For each example, copy the code into `file.c` and compile at optimization level `O2` to enable auto-vectorization:

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

The vectorized assembly code contains the Advanced SIMD instructions, for example `vld1`, `vshl`, and `vst1`. These Advanced SIMD instructions are not generated when compiling the example with the non-vectorizable loop.

Table 4-7 Assembly code from vectorizable and non-vectorizable loops

Vectorized assembly code

```
DoubleBuffer1:
.fnstart
@ BB#0:
        movw    r0, :lower16:buffer
        movt    r0, :upper16:buffer
        vld1.64 {d16, d17}, [r0:128]
        mov     r1, r0
        vshl.i32        q8, q8, #1
        vst1.32 {d16, d17}, [r1:128]!
        vld1.64 {d16, d17}, [r1:128]
        vshl.i32        q8, q8, #1
        vst1.64 {d16, d17}, [r1:128]
        add     r1, r0, #32
        vld1.64 {d16, d17}, [r1:128]
        vshl.i32        q8, q8, #1
        vst1.64 {d16, d17}, [r1:128]
        add     r1, r0, #48
        vld1.64 {d16, d17}, [r1:128]
        vshl.i32        q8, q8, #1
        vst1.64 {d16, d17}, [r1:128]
        add     r1, r0, #64
        add     r0, r0, #80
        vld1.64 {d16, d17}, [r1:128]
        vshl.i32        q8, q8, #1
        vst1.64 {d16, d17}, [r1:128]
        vld1.64 {d16, d17}, [r0:128]
        vshl.i32        q8, q8, #1
        vst1.64 {d16, d17}, [r0:128]
        bxlr
```

Non-vectorized assembly code

```
DoubleBuffer2:
    .fnstart
@ BB#0:
           movw    r0, :lower16:buffer
        movt    r0, :upper16:buffer
        ldr     r1, [r0]
        lsl     r1, r1, #1
        str     r1, [r0]
        ldr     r1, [r0, #12]
        lsl     r1, r1, #1
        str     r1, [r0, #12]
        ldr     r1, [r0, #24]
        lsl     r1, r1, #1
        str     r1, [r0, #24]
        ldr     r1, [r0, #36]
        lsl     r1, r1, #1
        str     r1, [r0, #36]
        ldr     r1, [r0, #48]
        lsl     r1, r1, #1
        str     r1, [r0, #48]
        ldr     r1, [r0, #60]
        lsl     r1, r1, #1
        str     r1, [r0, #60]
        ldr     r1, [r0, #72]
        lsl     r1, r1, #1
        str     r1, [r0, #72]
        ldr     r1, [r0, #84]
        lsl     r1, r1, #1
        str     r1, [r0, #84]
        ldr     r1, [r0, #4]
        lsl     r1, r1, #1
        str     r1, [r0, #4]
        ldr     r1, [r0, #16]
        lsl     r1, r1, #1
        ...
        bx    lr
```

Note

Advanced SIMD (Single Instruction Multiple Data), 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 ACLE Q1 2019*, *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 file.c -S -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 file.c -S -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 table shows two sample implementations of a routine to calculate `n!` that together illustrate loop termination overhead. The first implementation calculates `n!` using an incrementing loop, while the second routine calculates `n!` using a decrementing loop.

Table 4-8 C code for incrementing and decrementing loops

Incrementing loop	Decrementing loop

`int fact1(int n)`
`{`
`    int i, fact = 1;`
`    for (i = 1; i <= n; i++)`
`        fact *= i;`
`    return (fact);`
`}`
	`int fact2(int n)`
	`{`
	`    unsigned int i, fact = 1;`
	`    for (i = n; i != 0; i--)`
	`        fact *= i;`
	`    return (fact);`
	`}`

The following table shows the corresponding disassembly for each of the preceding sample implementations. Generate the disassembly using:

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

Table 4-9 C disassembly for incrementing and decrementing loops

Incrementing loop	Decrementing loop

`fact1:`
`        mov     r1, r0`
`        mov     r0, #1`
`        cmp     r1, #1`
`        bxlt    lr`
`        mov     r2, #0`
`.LBB0_1:`
`        add     r2, r2, #1`
`        mul     r0, r0, r2`
`        cmp     r1, r2`
`        bne     .LBB0_1`
`        bx      lr`
	`fact2:`
	`        mov     r1, r0`
	`        mov     r0, #1`
	`        cmp     r1, #0`
	`        bxeq    lr`
	`.LBB1_1:`
	`        mul     r0, r0, r1`
	`        subs    r1, r1, #1`
	`        bne     .LBB1_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, 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, Arm recommends writing infinite loops in the following way:

```
void infinite_loop(void) {
  while (1)
    asm volatile("");    // this line is considered to have side-effects
}
```

`armclang` does not delete or move the loop, because it has side-effects.

Related information

-O (armclang)

pragma unroll

-fvectorize (armclang)

4.3 Inlining functions

Arm Compiler 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 4-10 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 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 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 is equivalent to `__declspec(noinline)`.
`-fno-inline-functions`	This is a compiler command-line option. Specify this option to the compiler to disable inlining. This option overrides the `__inline__` hint.

Note

- Arm Compiler 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* on page 4-86 in the *Software Development Guide*.
- 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 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
```

The generated code shows inlining:

Table 4-11 Effect of -fno-inline-functions

Compiling with -fno-inline-functions

```
foo:                                    @ @foo
        .fnstart
@ BB#0:
        .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}
.Lfunc_end0:
       .size   foo, .Lfunc_end0-foo
        .cantunwind
        .fnend
```

Compiling without -fno-inline-functions

```
foo:                                    @ @foo
        .fnstart
@ BB#0:
        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
.Lfunc_end0:
       .size   foo, .Lfunc_end0-foo
        .cantunwind
        .fnend
```

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`.

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 *Arm® Architecture Procedure Call Standard* (AAPCS) or the *Arm® Architecture 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 AAPCS64 requires 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 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 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 provides a pragma and attribute to pack the members in a structure or union without any padding.

Table 4-12 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`.

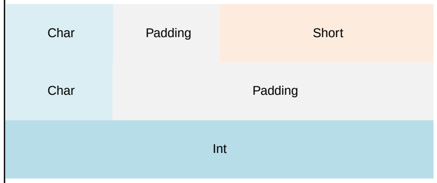
```
armlink file.o --info=sizes
```

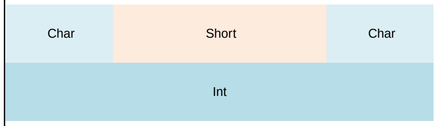
The linker output shows the total memory used by the two objects `c` and `d`. For example:

```
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
```

Table 4-13 Packing structures

Code	Packing	Size of structure

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

```
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**	8. The alignment of the structure is 1 byte.

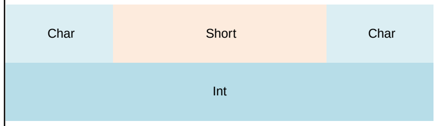
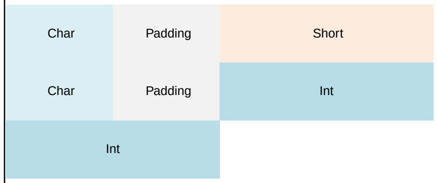
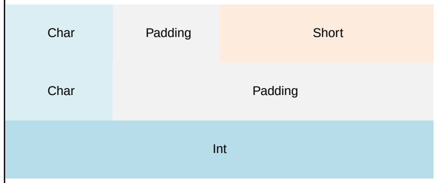
```
#pragma pack (1)
struct stc
{
    char one;
    short two;
    char three;
    int four;
} c,d;

int main (void)
{
    c.one=1;
    return 0;
}
```
	 **Figure 4-3 Structure with pragma pack with 1 byte alignment**	8. The alignment of the structure is 1 byte.

Table 4-13 Packing structures (continued)

Code	Packing	Size of structure

```
#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	10. The alignment of the structure is 2 bytes.

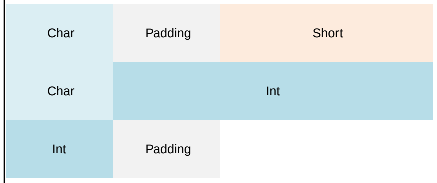
```
#pragma pack (4)
struct stc
{
    char one;
    short two;
    char three;
    int four;
} c,d;

int main (void)
{
    c.one=1;
    return 0;
}
```
	 Figure 4-5 Structure with pragma pack with 4 byte alignment	12. The alignment of the structure is 4 bytes.

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))`.

Table 4-14 Packing individual members

Code	Packing	Size

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

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)) foobar
{
  char x;
  short y;
};

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

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

Note

This topic includes descriptions of [ALPHA] features. See *Support level definitions* on page Appx-A-262.

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 | -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` along with other aggressive optimizations that might violate strict compliance with language standards.

`-Omax`
: Enables all the optimizations from `-Ofast` along 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:

- Link-Time Optimization 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 levels*.

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 [ALPHA]`
: C++ only. Disables the generation of code that is required to support Run Time Type Information (RTTI) features.

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

`-flto`

Enables Link-Time Optimization (LTO), which enables the linker to make additional optimizations across multiple source files. See *4.8 Optimizing across modules with Link-Time Optimization* on page 4-86 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.

In addition, 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`). 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.

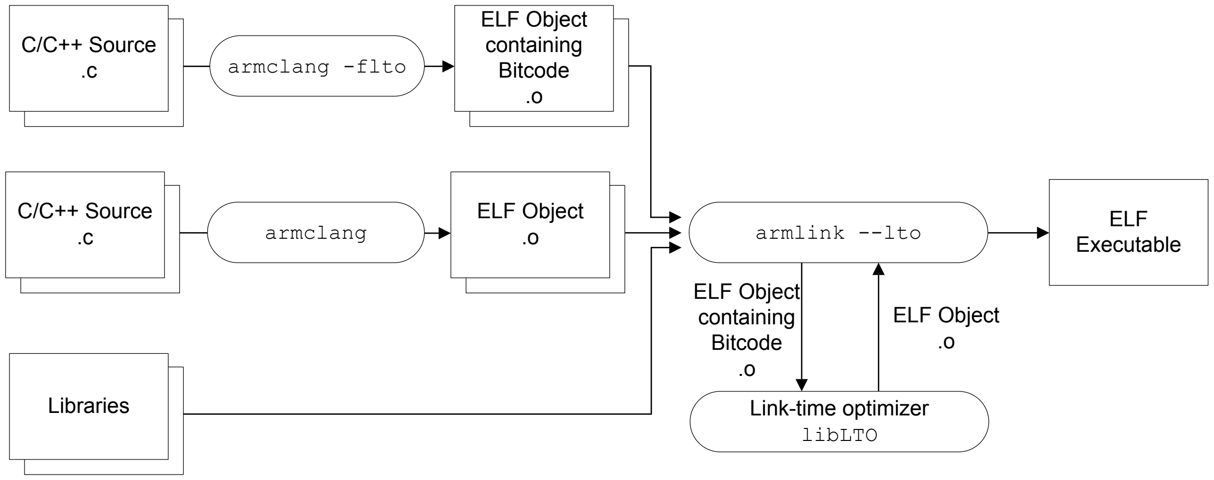


Figure 4-7 Link-Time Optimization

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 `.llvmbc` that holds LLVM bitcode.

Section `.llvmbc` is reserved. You must not create an `.llvmbc` section with, 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.

This section contains the following subsections:
- *4.8.1 Enabling Link-Time Optimization* on page 4-86.
- *4.8.2 Restrictions with Link-Time Optimization* on page 4-88.
- *4.8.3 Removing unused code across multiple object files* on page 4-88.

4.8.1 Enabling Link-Time Optimization

You must enable *Link-Time Optimization* (LTO) in both `armclang` and `armlink`.

To enable LTO:

1. At compilation time, use the `armclang` option `-flto` to produce ELF files suitable for LTO. These ELF files contain bitcode in a `.llvmbc` section.

   Note

   The `armclang` option `-Omax` automatically enables 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 1: 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 2: 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.

4.8.2 Restrictions with Link-Time Optimization

Link-Time Optimization (LTO) has a few restrictions in Arm Compiler 6. Future releases might have fewer restrictions and more features. The user interface to LTO might change in future releases.

No bitcode libraries

`armlink` only supports bitcode objects on the command line. It does not support bitcode objects coming from libraries. `armlink` gives an error message if it encounters a file containing bitcode while loading from a library.

Although `armar` silently accepts ELF files that are produced with `armclang -flto`, these files currently do not have a proper symbol table. Therefore, the generated archive has incorrect index information and `armlink` cannot find any symbols in this archive.

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.

Arm recommends that LTO is only performed on code and data that does not require precise placement in the scatter file, with general Input section selectors such as `*(+RO)` and `.ANY(+RO)` used to select sections that LTO generates.

It is not possible to match bitcode in `.llvmbc` sections by name in a scatter file.

Note

The scatter-loading interface is subject to change in future versions of Arm Compiler 6.

Executable and library compatibility

The `armclang` executable and the `libLTO` library must come from the same Arm Compiler 6 installation. Any use of `libLTO` other than the library supplied with Arm Compiler 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.
- LTO does not honor `armclang` options `-ffunction-sections` and `-fdata-sections`.
- LTO does not honor the `armclang -mexecute-only` option. If you use the `armclang -flto` or `-Omax` options, then the compiler cannot generate execute-only code.
- LTO does not work correctly when two bitcode files are compiled for different targets.

4.8.3 Removing unused code across multiple object files

Link-time optimization (LTO) can remove unused code across multiple object files, particularly when the code contains conditional calls to functions that are otherwise unused.

In this example:

- The function `main()` calls an externally defined function `foo()`, and returns the value that `foo()` returns. Because this function is externally defined, the compiler cannot inline or otherwise optimize it when compiling `main.c`, without using LTO.
- The file `foo.c` contains the following functions:

  `foo()`

  If the parameter `a` is nonzero, `foo()` conditionally calls a function `bar()`.

  `bar()`

  This function prints a message.

In this case, `foo()` is called with the parameter `a == 0`, so `bar()` is not called at run time.

Example code that is used in the following procedure:

```
// main.c
extern int foo(int a);

int main(void)
{
    return foo(0);
}
```

```
// foo.c
#include <stdio.h>

int foo(int a);
void bar(void);

/* `foo()` conditionally calls `bar()`
   depending on the value of `a`
 */
int foo(int a)
{
    if (a == 0)
    {
        return 0;
    }
    else
    {
        bar();

        return 0;
    }
}

void bar(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 foo.c -o foo.o
   armlink main.o foo.o -o image_without_lto.axf
   fromelf --text -c -z image_without_lto.axf
   ```

   Results:

   The compiler cannot inline the call to `foo()` because it is in a different object from `main()`. Therefore, the compiler must keep the conditional call to `bar()` within `foo()`, because the compiler does not have any information about the value of the parameter `a` while `foo.c` is being compiled:

   ```
   $a.0
   foo
       0x00008bd8:    e3500000    ..P.    CMP      r0,#0
       0x00008bdc:    0a000004    ....    BEQ      0x8bf4 ; foo + 28
       0x00008be0:    e92d4800    .H-.    PUSH     {r11,lr}
       0x00008be4:    e3080c44    D...    MOV      r0,#0x8c44
       0x00008be8:    e3400000    ..@.    MOVT     r0,#0
       0x00008bec:    fafffd28    (...    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        foo ; 0x8bd8
```

Also, `bar()` uses the Arm C library function `printf()`. In this example, `printf()` is optimized to `puts()` and inlined into `foo()`. 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

      3128        200         44         16        348       1740   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 foo.c -o foo.o
armlink --lto main.o foo.o -o image_with_lto.axf
fromelf --text -c -z image_with_lto.axf
```

Results:

Although the compiler does not have any information about the call to `foo()` from `main()` when compiling `foo.c`, at link time, it is known that:

- `foo()` is only ever called once, with the parameter `a == 0`.
- `bar()` 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 `foo()` into `main()`.
- Removing the code to conditionally call `bar()` from `foo()` 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
```

Related references

4.6 Optimizing for code size or performance on page 4-83

4.8 Optimizing across modules with Link-Time Optimization on page 4-86

4.10 How optimization affects the debug experience on page 4-96

Related information

-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`:

`foo.c`:

```
#include <stdio.h>

const char foo_string1[] = "Hello from foo_A!()\n";
const char foo_string2[] = "Hello from foo_B!()\n";

void foo_A(void)
{
    printf("%s", foo_string1);
}

void foo_B(void)
{
    printf("%s", foo_string2);
}
```

`bar.c`:

```
#include <stdio.h>

const char bar_string1[] = "Hello from bar_A!()\n";
const char bar_string2[] = "Hello from bar_B!()\n";

void bar_A(void)
{
    printf("%s", bar_string1);
}

void bar_B(void)
{
    printf("%s", bar_string2);
}
```

`main.c`:

```
extern void foo_A(void);
extern void foo_B(void);
extern void bar_A(void);
extern void bar_B(void);
```

```
int main(void)
{
    foo_A();
    foo_B();
    bar_A();
    bar_B();

    return 0;
}
```

`scatter.sct`:

```
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 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.sct 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: 0x00000038, Max: 0xffffffff, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00001000   0x0000001c   Code   RO            3    .text.foo_A         foo.o
0x0000101c   0x0000001c   Code   RO            5    .text.foo_B         foo.o


Execution Region EXEC_BAR (Base: 0x00001400, Size: 0x00000038, Max: 0xffffffff, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00001400   0x0000001c   Code   RO            8    .text.bar_A         bar.o
0x0000141c   0x0000001c   Code   RO           10    .text.bar_B         bar.o
```

Example: Building with LTO enabled

Build the example code with LTO enabled:

```
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.sct foo.o bar.o main.o -o image.axf --lto --map --list=image.lst
```

The linker reports:

```
"scatter.sct", line 10 (column 16): Warning: L6314W: No section matches pattern foo.o(RO).
"scatter.sct", 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 -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.sct foo.o bar.o main.o -o image.axf --lto --map --list=image.lst
```

The linker still reports:

```
"scatter.sct", line 10 (column 16): Warning: L6314W: No section matches pattern foo.o(RO).
"scatter.sct", 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 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: 0x00000da8, Max: 0xffffffff, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object
...
0x00000b60   0x0000001c   Code   RO          397    .text.bar_A         lto-llvm-3d77ff.o
0x00000b7c   0x0000001c   Code   RO          399    .text.bar_B         lto-llvm-3d77ff.o
0x00000b98   0x0000001c   Code   RO          393    .text.foo_A         lto-llvm-3d77ff.o
0x00000bb4   0x0000001c   Code   RO          395    .text.foo_B         lto-llvm-3d77ff.o
```

`lto_llvm_3d77ff.o` is the LTO intermediate filename that the linker generates. You can change this name using the `armlink --lto_intermediate_filename` command-line option, though that does not help in this use case. Instead, section names must be used.

Example: Using section names for all functions within a C language source file

The easiest way to specify section names for all functions within a C language source file is to use `#pragma clang section`. For this example, rewrite the example code `foo.c` and `bar.c` as follows:

`foo.c`:

```
#include <stdio.h>

#pragma clang section text="foo_rotext" rodata="foo_rodata"

const char foo_string1[] = "Hello from foo_A!()\n";
const char foo_string2[] = "Hello from foo_B!()\n";

void foo_A(void)
{
    printf("%s", foo_string1);
}

void foo_B(void)
{
    printf("%s", foo_string2);
}
```

`bar.c`:

```
#include <stdio.h>

#pragma clang section text="bar_rotext" rodata="bar_rodata"

const char bar_string1[] = "Hello from bar_A!()\n";
const char bar_string2[] = "Hello from bar_B!()\n";

void bar_A(void)
{
    printf("%s", bar_string1);
}

void bar_B(void)
{
    printf("%s", bar_string2);
}
```

`#pragma clang section text="foo_rotext" rodata="foo_rodata"` specifies that code and read-only data (such as the string constants used within the calls to `printf()` in `foo.c`) are placed in named sections:

- `foo_rotext` for the code that is generated.
- `foo_rodata` for the read-only data that is generated.

Similar names are specified in `bar.c` for the code and data generated by that file. You can rewrite `scatter.sct` to use these section names as follows:

`scatter.sct`:

```
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 -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.sct foo.o bar.o main.o -o image.axf --lto --map --list=image.lst
```

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: 0x00000038, Max: 0xffffffff, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00001000   0x00000028   Code   RO          435    foo_rotext          lto-llvm-4392b4.o
```

```
0x00001028   0x0000002a   Data   RO          441    foo_rodata          lto-llvm-4392b4.o

Execution Region EXEC_BAR (Base: 0x00001400, Size: 0x00000038, Max: 0xffffffff, ABSOLUTE)

Base Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00001400   0x00000028   Code   RO          437    bar_rotext          lto-llvm-4392b4.o
0x00001428   0x0000002a   Data   RO          442    bar_rodata          lto-llvm-4392b4.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_RO` and `bar_RO` 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   : 7296 bytes (alignment 4)
    String table #9 '.strtab'
    Last local symbol no. 309

    Symbol table .symtab (455 symbols, 309 local)

      #  Symbol Name                Value      Bind  Sec  Type  Vis  Size
    ========================================================================
    ...
    297  foo_A                      0x00001000   Lc    4  Code  De   0x14
    298  foo_B                      0x00001014   Lc    4  Code  De   0x14
    299  bar_A                      0x00001400   Lc    5  Code  De   0x14
    300  bar_B                      0x00001414   Lc    5  Code  De   0x14
    301  foo_string1                0x00001028   Lc    4  Data  De   0x15
    302  foo_string2                0x0000103d   Lc    4  Data  De   0x15
    303  bar_string1                0x00001428   Lc    5  Data  De   0x15
    304  bar_string2                0x0000143d   Lc    5  Data  De   0x15
```

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 char[] constants.

Other considerations

Consider the following approaches:

- 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 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 the attribute `__attribute__((noinline))` must be used on a per-function basis. This approach can help achieve a better balance between explicit code placement and cross-file function inlining optimizations.

Related references

4.8 Optimizing across modules with Link-Time Optimization on page 4-86

Related information

-fno-inline-functions (armclang)

-flto (armclang)

-O (armclang)

__attribute__((noinline)) function 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, Arm recommends `-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*.

Chapter 5
Assembling Assembly Code

Describes how to assemble assembly source code with `armclang` and `armasm`.

It contains the following sections:

- *5.1 Assembling armasm and GNU syntax assembly code* on page 5-98.
- *5.2 Preprocessing assembly code* on page 5-100.

5.1 Assembling armasm and GNU syntax assembly code

The Arm Compiler toolchain can assemble both `armasm` and GNU syntax assembly language source code.

`armasm` and GNU are two different syntaxes for assembly language source code. They are similar, but have a number of differences. For example, `armasm` syntax identifies labels by their position at the start of a line, while GNU syntax identifies them by the presence of a colon.

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 `armasm` syntax and GNU syntax assembly to help you migrate legacy assembly code.

The following examples show equivalent `armasm` and GNU syntax assembly code for incrementing a register in a loop.

`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
```

GNU assembler syntax:

```
// Simple GNU syntax example
//
// Iterate round a loop 10 times, adding 1 to a register each time.

        .section .text,"ax"
        .balign 4

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
```

Related information

GNU Binutils - Using as

Migrating armasm syntax assembly code to GNU syntax

5.2 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 -D` command-line option.
- 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)

Chapter 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 which C or C++ provides.

For example:

- To access features which 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 is replaced during compilation by a specific sequence of low-level instructions.

  Note

  Arm intrinsics are recognized by Arm compilers, but 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.

It contains the following sections:

- *6.1 Using intrinsics* on page 6-103.
- *6.2 Writing inline assembly code* on page 6-105.

- *6.3 Calling assembly functions from C and C++* on page 6-108.

6.1 Using intrinsics

Compiler intrinsics are special functions whose implementations are known to the compiler. They 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 a wide variety of tasks but they do not provide built-in support for specific areas of application, for example *Digital Signal Processing* (DSP).

Within 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 than in C or C++. A typical example is the saturated add of two 32-bit signed two's complement integers, commonly used in DSP programming. The following example shows a C implementation of a saturated add operation:

```
#include <limits.h>
int L_add(const int a, const int b)
{
    int c;
    c = a + b;
    if (((a ^ b) & INT_MIN) == 0)
    {
        if ((c ^ a) & INT_MIN)
        {
            c = (a < 0) ? INT_MIN : INT_MAX;
        }
    }
    return c;
}
```

Using compiler intrinsics, you can achieve more complete coverage of target architecture instructions than you would from the instruction selection of the compiler.

An intrinsic function has the appearance of a function call in C or C++, but is replaced during compilation by a specific sequence of low-level instructions. The following example shows how to access the `__qadd` saturated add intrinsic:

```
#include <arm_acle.h>    /* Include ACLE intrinsics */

int foo(int a, int b)
{
  return __qadd(a, b);  /* Saturated add of a and b */
}
```

Using compiler intrinsics offers a number of performance benefits:

- The low-level instructions substituted for an intrinsic are either as efficient or more efficient than corresponding implementations in C or C++. This 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 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.

Note

If you need to use SVE and SVE2 intrinsics, see *7.2 Using SVE and SVE2 intrinsics directly in your C code* on page 7-116 for more information.

Related information

Compiler-specific intrinsics

ACLE support

NEON Programmer's Guide

6.2 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.

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
       : output_operand_list
      [: input_operand_list
      [: clobbered_register_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 may otherwise lead to the compiler removing the code block. The `volatile` qualifier is optional, but you should consider using it around your assembly code blocks to ensure the compiler does not remove them when compiling with `-O1` or above.

`code` is the assembly instruction, for example `"ADD R0, R1, R2"`. *`code_template`* is 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 *`output_operand_list`* before specifying the optional *`input_operand_list`* and *`clobbered_register_list`*.

`output_operand_list` is 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)
  );
```

input_operand_list is 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.

clobbered_register_list is an optional list of clobbered registers whose contents are not preserved. The list can be empty. In addition to registers, the list can also contain special arguments:

"cc"
: The instruction affects the condition code flags.

"memory"
: The instruction accesses unknown memory addresses.

The registers in *clobbered_register_list* must use lowercase letters rather than uppercase letters. An example instruction with a *clobbered_register_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 Armv8-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 *__attribute__((naked))* in the *Arm Compiler Reference Guide*.

Related information

armclang Inline Assembler

Migrating armasm syntax assembly code to GNU syntax

Semihosting for AArch32 and AArch64

6.3 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 `.globl` and `.type`:

```
     .globl   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 *Procedure Call Standard for the Arm® Architecture (AAPCS)*.

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

Chapter 7
SVE Coding Considerations with Arm® Compiler

Describes best practices for writing code that uses the SVE and SVE2 features of Arm Compiler.

It contains the following sections:

- *7.1 Embedding SVE assembly code directly into C and C++ code* on page 7-111.
- *7.2 Using SVE and SVE2 intrinsics directly in your C code* on page 7-116.

7.1 Embedding SVE assembly code directly into C and C++ code

Inline assembly (or inline `asm`) provides a mechanism for inserting hand-written assembly instructions into C and C++ code. This 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.

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 compiler supports the GNU form of inline assembly. Note that it does not support the Microsoft form of inline assembly.

More detailed documentation of the `asm` construct is available at *the GCC website*.

Inline assembly statements have the following form:

`asm ("`*`instructions`*`" : `*`outputs`*` : `*`inputs`*` : `*`side-effects`*`);`

Where:

`instructions`
is a text string that contains AArch64 assembly instructions, with at least one newline sequence \n between consecutive instructions.

`outputs`
is a comma-separated list of outputs from the assembly instructions.

`inputs`
is a comma-separated list of inputs to the assembly instructions.

`side-effects`
is a comma-separated list of effects that the assembly instructions have, besides reading from inputs and writing to outputs.

Additionally, the `asm` keyword might need to be followed by the `volatile` keyword.

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 specifies that the assembly instructions might read from one of the inputs (specified in the `asm` statement's inputs section) after writing to the output.

The second form has the register class preceded by =. This 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`
: if the register for this output when used within the assembly instructions is a general-purpose register (x0-x30)

`w`
: if 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]`
: to refer to an r-class output as `x`*N* or a w-class output as `v`*N*

`%w[name]`
: to refer to an r-class output as `w`*N*

`%s[name]`
: to refer to a w-class output as `s`*N*

`%d[name]`
: to refer 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`
: if the input is to be placed in a general-purpose register (x0-x30)

`w`
: if the input is to be placed in a SIMD and floating-point register (v0-v31).

[*output-name*]
: if the input is to be placed in the same register as output *output-name*. 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 the Outputs section above (%[*name*], %w[*name*], %s[*name*], %d[*name*]) to refer to both the input and the output.

i
: if 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 (%[*name*], %w[*name*], %s[*name*], %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 *side-effects* section of an asm statement tells the compiler what these additional effects are. Each entry must be one of the following:

"memory"
: if the asm statement reads from or writes to memory. This is necessary even if inputs contain pointers to the affected memory.

"cc"
: if the asm statement modifies the condition-code flags.

"x*N*"
: if the asm statement modifies general-purpose register *N*.

"v*N*"
: if the asm statement modifies SIMD and floating-point register *N*.

"z*N*"
: if 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*"
: if 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` to refer the next definition (`f` is for forward) 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. This allows the compiler output to contain many copies of this code without creating any ambiguity.

```
    ...pre-loop test...
    b.none      2f
1:
    ...loop...
    b.any       1b
2:
```

Example

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.

7.2 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 for SVE* 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 for SVE* 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 Arm C Language Extensions 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.

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	...

Table 7-1 Element selection by predicate type svbool_t (continued)

Vector type	Element selected by each svbool_t bit									
`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 for SVE* 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.

`_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 for SVE* 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` `-march` or `-mcpu` options. 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 Reference Guide* for more information.

Example – Naïve step-1 daxpy

daxpy is a BLAS (Basic Linear Algebra Subroutines) subroutine 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 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]
}
```

The following steps explain this example:

[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]
}
```

The following steps explain this example:

[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.

Chapter 8
Mapping Code and Data to the Target

There are various options in Arm Compiler to control how code, data and other sections of the image are mapped to specific locations on the target.

It contains the following sections:

- *8.1 What the linker does to create an image* on page 8-123.
- *8.2 Placing data items for target peripherals with a scatter file* on page 8-125.
- *8.3 Placing the stack and heap with a scatter file* on page 8-126.
- *8.4 Root region* on page 8-127.
- *8.5 Placing functions and data in a named section* on page 8-130.
- *8.6 Placing functions and data at specific addresses* on page 8-132.
- *8.7 Bare-metal Position Independent Executables* on page 8-140.
- *8.8 Placement of Arm® C and C++ library code* on page 8-142.
- *8.9 Placement of unassigned sections* on page 8-144.
- *8.10 Placing veneers with a scatter file* on page 8-154.
- *8.11 Preprocessing a scatter file* on page 8-155.
- *8.12 Reserving an empty block of memory* on page 8-157.
- *8.13 Aligning regions to page boundaries* on page 8-159.
- *8.14 Aligning execution regions and input sections* on page 8-160.

8.1 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 Armv7-M or Armv8-M architectures.

This section contains the following subsection:

- *8.1.1 What you can control with a scatter file* on page 8-123.

8.1.1 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.

———————————

8.2 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.

To make sure that the data items are placed at the correct address for the peripherals, use the `__attribute__((section(".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(".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
    {
        *(.ARM.__at_0x10000000)
    }
}
```

3. Build the image.

```
armclang --target=arm-arm-eabi-none -mcpu=cortex-a9 peripheral.c -g -c -o peripheral.o
armlink --cpu=cortex-a9 --scatter=scatter.scat --map --symbols peripheral.o --
output=peripheral.axf > map.txt
```

Results: The memory map for load region `LR_3` is:

```
  Load Region LR_3 (Base: 0x10000000, Size: 0x00000004, 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   Data   RW           5    .ARM.__at_0x10000000  peripheral.o
```

8.3 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.

———— **Note** ————

- If you re-implement `__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.

   Example:

   ```
   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`.

8.4 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
}
```

This section contains the following subsections:
- *8.4.1 Effect of the ABSOLUTE attribute on a root region* on page 8-127.
- *8.4.2 Effect of the FIXED attribute on a root region* on page 8-128.

8.4.1 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
}
```

8.4.2 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:

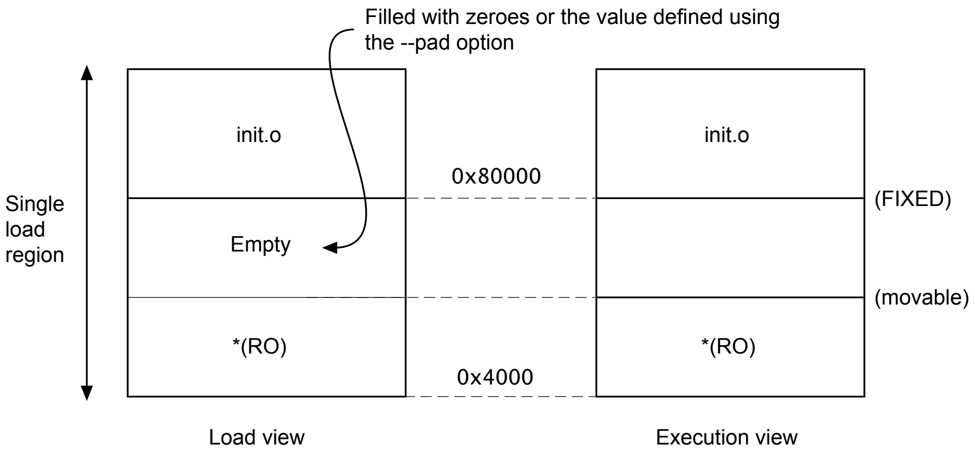


Figure 8-1 Memory map for fixed execution regions

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
}
```

8.5 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")))`.

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-eabi-none -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.

Example:

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 *8.6 Placing functions and data at specific addresses* on page 8-132.

Related information

Semihosting for AArch32 and AArch64

8.6 Placing functions and data at specific addresses

To 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 rest of the input files.

This section contains the following subsections:

- *8.6.1 Placing __at sections at a specific address* on page 8-132.
- *8.6.2 Restrictions on placing __at sections* on page 8-132.
- *8.6.3 Automatically placing __at sections* on page 8-133.
- *8.6.4 Manually placing __at sections* on page 8-134.
- *8.6.5 Placing a key in flash memory with an __at section* on page 8-134.
- *8.6.6 Placing constants at fixed locations* on page 8-135.
- *8.6.7 Placing jump tables in ROM* on page 8-136.
- *8.6.8 Placing a variable at a specific address without scatter-loading* on page 8-137.
- *8.6.9 Placing a variable at a specific address with scatter-loading* on page 8-138.

8.6.1 Placing __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.

8.6.2 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 not be used in *Base Platform Application Binary Interface* (BPABI) executables and BPABI *dynamically linked libraries* (DLLs).
- `__at` sections must have an address that is a multiple of their alignment.
- `__at` sections ignore any `+FIRST` or `+LAST` ordering constraints.

8.6.3 Automatically placing __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 and execution region for the `__at` section.

All linker execution regions created by `--autoat` have the `UNINIT` scatter-loading attribute. If you require a 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 and execution region for the __at section
; .ARM.__at_0x8000 because it lies outside all candidate regions.
```

8.6.4 Manually placing __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
    }
}
```

8.6.5 Placing 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
    }
}
```

8.6.6 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-eabi-none -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
```

Results: 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.
   Results: 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                                        ....
```

8.6.7 Placing jump tables in ROM

You might find that jump tables are placed in RAM rather than in ROM.

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-eabi-none -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
```

   Results: 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:    eaffffff    ....    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                ............
...
```

8.6.8 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
```

8.6.9 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

8.7 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

- Bare-metal PIE support is deprecated.
- There is support for `-fropi` and `-frwpi` in `armclang`. 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

Consider the following simple example 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 `armclang -fbare-metal-pie` and link with `armlink --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 Position Independent Executable.

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, use the `armlink --lto_relocation_model=pic` option 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:

- AArch32 state only.
- 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.
- When writing assembly code for position independence, some instructions (`LDR`, for example) let you specify a PC-relative address in the form of a label. For example:

  ```
  LDR r0,=__main
  ```

  This 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
  ...
      AREA WRITE_TEST, DATA, READWRITE
  __main_addr DCD __main
      END
  ```

Using a scatter file

An example scatter file is:

```
LR 0x0 PI
{
    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
}
```

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_`, so it is necessary to ensure that this routine only uses PC relative addressing.

Related information

--fpic (armlink)

--pie (armlink)

--bare_metal_pie (armlink)

--ref_pre_init (armlink)

-fbare-metal-pie (armclang)

-fropi (armclang)

-frwpi (armclang)

8.8 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.

———————————

This section contains the following subsections:
- *8.8.1 Placing code in a root region* on page 8-142.
- *8.8.2 Placing Arm® C library code* on page 8-142.
- *8.8.3 Placing Arm® C++ library code* on page 8-143.

8.8.1 Placing 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
  }
}
```

8.8.2 Placing 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`.

8.8.3 Placing 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.

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.

8.9 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.

This section contains the following subsections:

- *8.9.1 Default rules for placing unassigned sections* on page 8-144.
- *8.9.2 Command-line options for controlling the placement of unassigned sections* on page 8-145.
- *8.9.3 Prioritizing the placement of unassigned sections* on page 8-145.
- *8.9.4 Specify the maximum region size permitted for placing unassigned sections* on page 8-146.
- *8.9.5 Examples of using placement algorithms for .ANY sections* on page 8-146.
- *8.9.6 Example of next_fit algorithm showing behavior of full regions, selectors, and priority* on page 8-148.
- *8.9.7 Examples of using sorting algorithms for .ANY sections* on page 8-149.
- *8.9.8 Behavior when .ANY sections overflow because of linker-generated content* on page 8-150.

8.9.1 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`.

8.9.2 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.

8.9.3 Prioritizing the placement of unassigned sections

You can give a priority ordering when placing unassigned sections with multiple `.ANY` module selectors.

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
    }
}
```

8.9.4 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.

8.9.5 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 8-1 Input section properties for placement of .ANY sections

Name	Size
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(+R0)` 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.

8.9.6 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 8-2 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.

8.9.7 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:

Table 8-3 Input section properties and ordering for sections_a.o and sections_b.o

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 8-4 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 8-5 Sort order for cmdline algorithm

Name	Size
seca_1	0x4
seca_2	0x4
seca_3	0x10
seca_4	0x14
secb_1	0x4
secb_2	0x4
secb_3	0x10
secb_4	0x14

That is, the input sections are sorted by command-line index.

8.9.8 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:

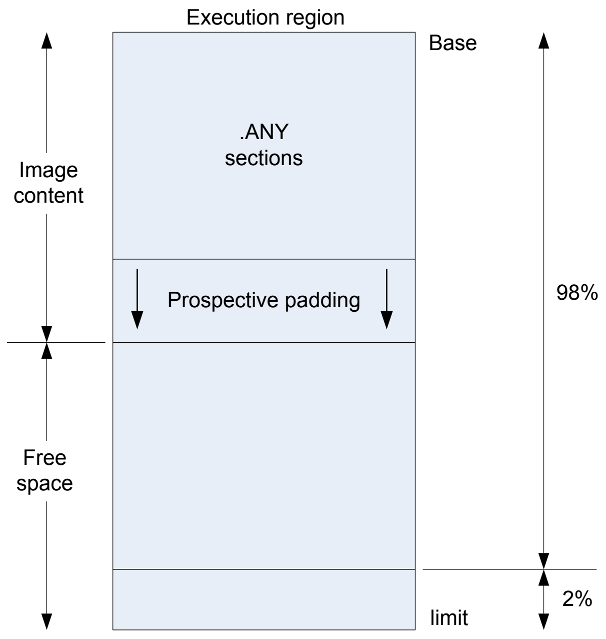


Figure 8-2 .ANY contingency

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)
```

8.10 Placing veneers with a scatter file

You can place veneers at a specific location with a linker-generated symbol.

Veneers allow switching between A32 and T32 code or allow a longer program jump than can be specified in a single instruction.

Procedure

1. 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.

8.11 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
```

This section contains the following subsections:

- *8.11.1 Default behavior for `armclang` -E in a scatter file* on page 8-155.
- *8.11.2 Using other preprocessors in a scatter file* on page 8-155.

8.11.1 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 relative `#include`s 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.

8.11.2 Using 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.

8.12 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.

This section contains the following subsections:

- *8.12.1 Characteristics of a reserved empty block of memory* on page 8-157.
- *8.12.2 Example of reserving an empty block of memory* on page 8-157.

8.12.1 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.

8.12.2 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.

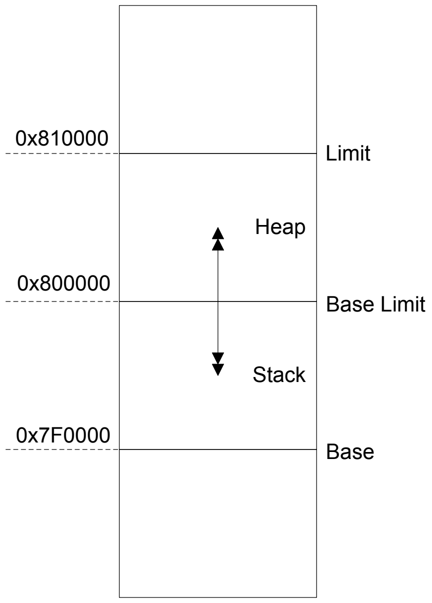


Figure 8-3 Reserving a region for the stack

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.

8.13 Aligning 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.

8.14 Aligning 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 OVERALIGN some sections where it might be unsafe to do so. For more information, see *Syntax of an input section description*.

Chapter 9
Overlays

Describes the Arm Compiler support for overlays to enable you to have multiple load regions at the same address.

It contains the following sections:

- *9.1 Overlay support in Arm® Compiler* on page 9-162.
- *9.2 Automatic overlay support* on page 9-163.
- *9.3 Manual overlay support* on page 9-168.

9.1 Overlay support in Arm® Compiler

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 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.

Related concepts

9.2 Automatic overlay support on page 9-163

9.3 Manual overlay support on page 9-168

Related information

__attribute__((section("name"))) function attribute

AREA

Execution region attributes

--emit_debug_overlay_section linker option

--overlay_veneers linker option

9.2 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.

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 --emit_debug_overlay_section` command-line options to add extra debug information to the image. This option permits an overlay-aware debugger to track which overlay is currently active.

This section contains the following subsections:

- *9.2.1 Automatically placing code sections in overlay regions* on page 9-163.
- *9.2.2 Overlay veneer* on page 9-164.
- *9.2.3 Overlay data tables* on page 9-165.
- *9.2.4 Limitations of automatic overlay support* on page 9-165.
- *9.2.5 Writing an overlay manager for automatically placed overlays* on page 9-166.

9.2.1 Automatically placing code sections in overlay regions

Arm Compiler can automatically place code sections into overlay regions.

You identify the sections in your code that are to become overlays by giving them names of the form `.ARM.overlayN`, 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.overlayN` 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.overlayN` 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
         .globl    foo
         .p2align    2
         .type    foo,%function
     foo:                                  @ @foo
         ...
         .fnend

         .section .ARM.overlay2,"ax",%progbits
         .globl    bar
     ```

```
    .p2align    2
    .type    bar,%function
bar:                                  @ @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 `armlink --emit_debug_overlay_section` command-line option.

Related information

__attribute__((section("name"))) function attribute

AREA

Execution region attributes

--emit_debug_overlay_section linker option

--overlay_veneers linker option

9.2.2 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

9.2.3 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 concepts

9.2 Automatic overlay support on page 9-163

9.2.4 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.overlayN`, `armlink` still treats them as different overlays. `armlink` assigns a different integer ID to each overlay.
- The `armlink --any_placement` command-line option 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 concepts

9.2 Automatic overlay support on page 9-163

9.2.5 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 such as cooperative or preemptive thread switching, coroutines, and setjmp/longjmp, unless it is customized to keep the parallel stack of the overlay manager consistent.

The `armlink --info=auto_overlay` option 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 `fromelf --bin` output. 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 concepts

9.2 Automatic overlay support on page 9-163

Related information

--info linker option

Related information

__attribute__((section("name"))) function attribute

AREA

Execution region attributes

--emit_debug_overlay_section linker option

--overlay_veneers linker option

9.3 Manual overlay support

To manually allocate code sections to overlay regions, you must set up a scatter file to locate the overlays.

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.

This section contains the following subsections:

- *9.3.1 Manually placing code sections in overlay regions* on page 9-168.
- *9.3.2 Writing an overlay manager for manually placed overlays* on page 9-169.

9.3.1 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 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 9-1 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

9.3.2 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.

The 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 downloadable example contains a `Readme.txt` file that describes details of each source file.

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, the two functions, `func1()` and `func2()`, and their corresponding data are placed in `CODE_ONE`, `CODE_TWO`, `DATA_ONE`, `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()` and `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 concepts

9.3 Manual overlay support on page 9-168

Related information

Use of $Super$$ and $Sub$$ to patch symbol definitions

Related concepts

9.1 Overlay support in Arm® Compiler on page 9-162

Related information

Execution region attributes

--emit_debug_overlay_relocs linker option

--emit_debug_overlay_section linker option

Chapter 10
Embedded Software Development

Describes how to develop embedded applications with Arm Compiler, with or without a target system present.

It contains the following sections:

- *10.1 About embedded software development* on page 10-177.
- *10.2 Default compilation tool behavior* on page 10-178.
- *10.3 C library structure* on page 10-179.
- *10.4 Default memory map* on page 10-180.
- *10.5 Application startup* on page 10-182.
- *10.6 Tailoring the C library to your target hardware* on page 10-183.
- *10.7 Reimplementing C library functions* on page 10-185.
- *10.8 Tailoring the image memory map to your target hardware* on page 10-187.
- *10.9 About the scatter-loading description syntax* on page 10-188.
- *10.10 Root regions* on page 10-189.
- *10.11 Placing the stack and heap* on page 10-190.
- *10.12 Run-time memory models* on page 10-191.
- *10.13 Reset and initialization* on page 10-193.
- *10.14 The vector table* on page 10-194.
- *10.15 ROM and RAM remapping* on page 10-195.
- *10.16 Local memory setup considerations* on page 10-196.
- *10.17 Stack pointer initialization* on page 10-197.
- *10.18 Hardware initialization* on page 10-198.
- *10.19 Execution mode considerations* on page 10-199.
- *10.20 Target hardware and the memory map* on page 10-200.
- *10.21 Execute-only memory* on page 10-201.
- *10.22 Building applications for execute-only memory* on page 10-202.

- *10.23 Vector table for ARMv6 and earlier, ARMv7-A and ARMv7-R profiles* on page 10-203.
- *10.24 Vector table for M-profile architectures* on page 10-204.
- *10.25 Vector Table Offset Register* on page 10-205.
- *10.26 Integer division-by-zero errors in C code* on page 10-206.

10.1 About 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 re-implement 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 C library initialization routines.

10.2 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

10.3 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.

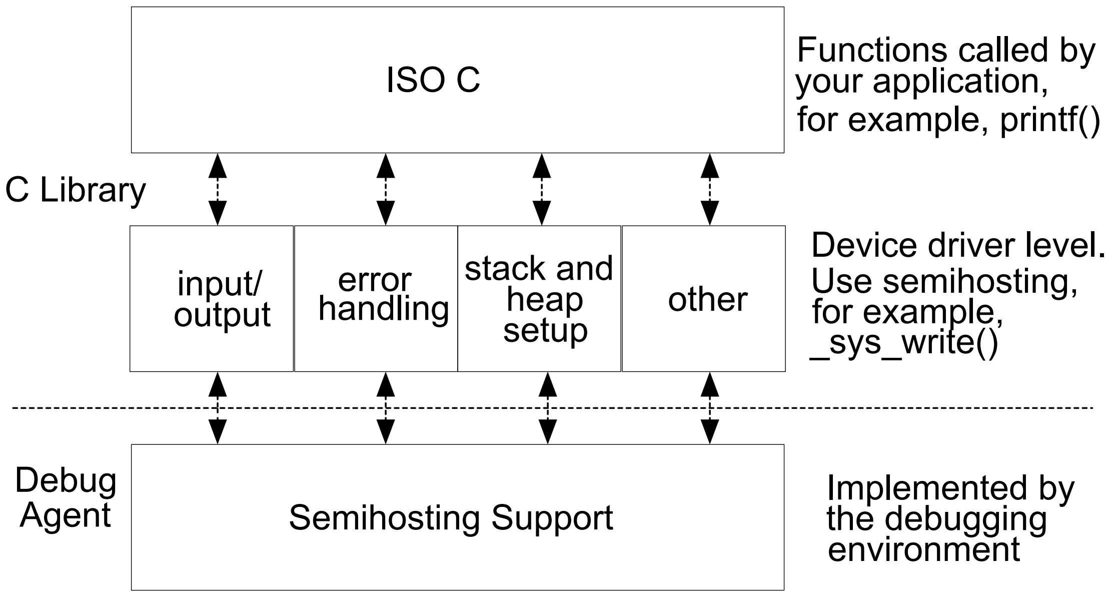

ISO C
Functions called by your application, for example, printf()
C Library
input/ output
error handling
stack and heap setup
other
Device driver level. Use semihosting, for example, _sys_write()
Debug Agent
Semihosting Support
Implemented by the debugging environment

Figure 10-1 C library structure

Related information
The Arm C and C++ libraries
The C and C++ library functions
Semihosting for AArch32 and AArch64

10.4 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.

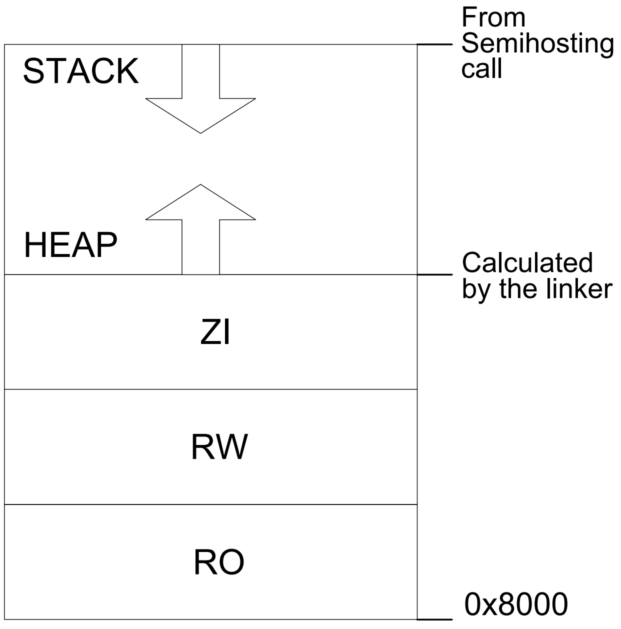


Figure 10-2 Default memory map

———— **Note** ————

The processors that are based on Armv6-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:

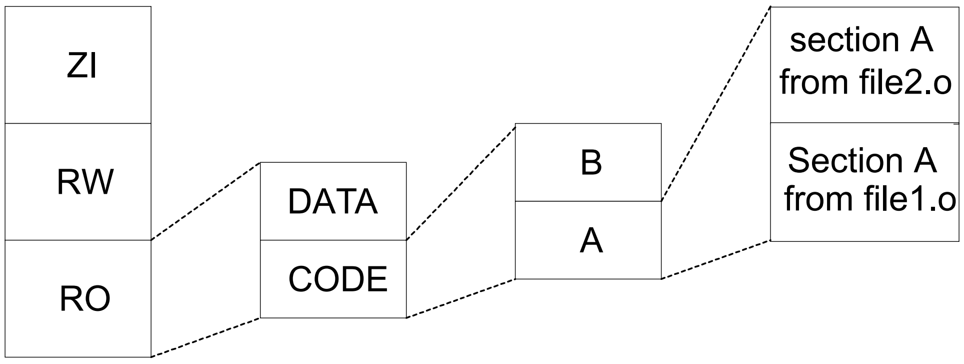


Figure 10-3 Linker placement rules

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 concepts

10.6 Tailoring the C library to your target hardware on page 10-183

Related information

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

10.5 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.

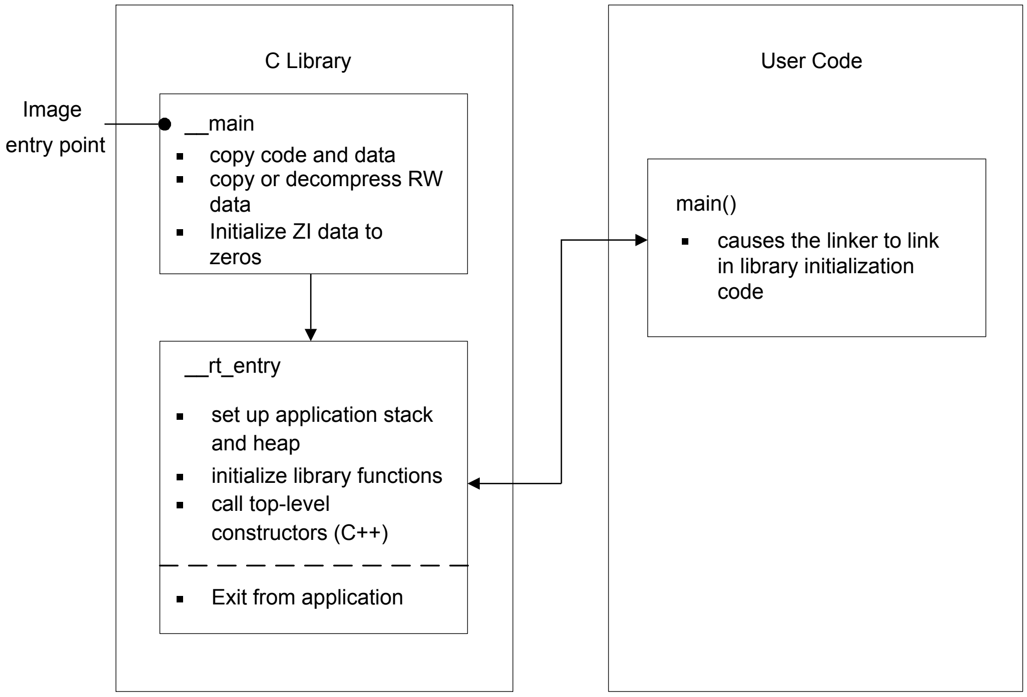

C Library
User Code
Image
entry point
__main
▪ copy code and data
▪ copy or decompress RW data
▪ Initialize ZI data to zeros
__rt_entry
▪ set up application stack and heap
▪ initialize library functions
▪ call top-level constructors (C++)
▪ Exit from application
main()
▪ causes the linker to link in library initialization code

Figure 10-4 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

10.6 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.

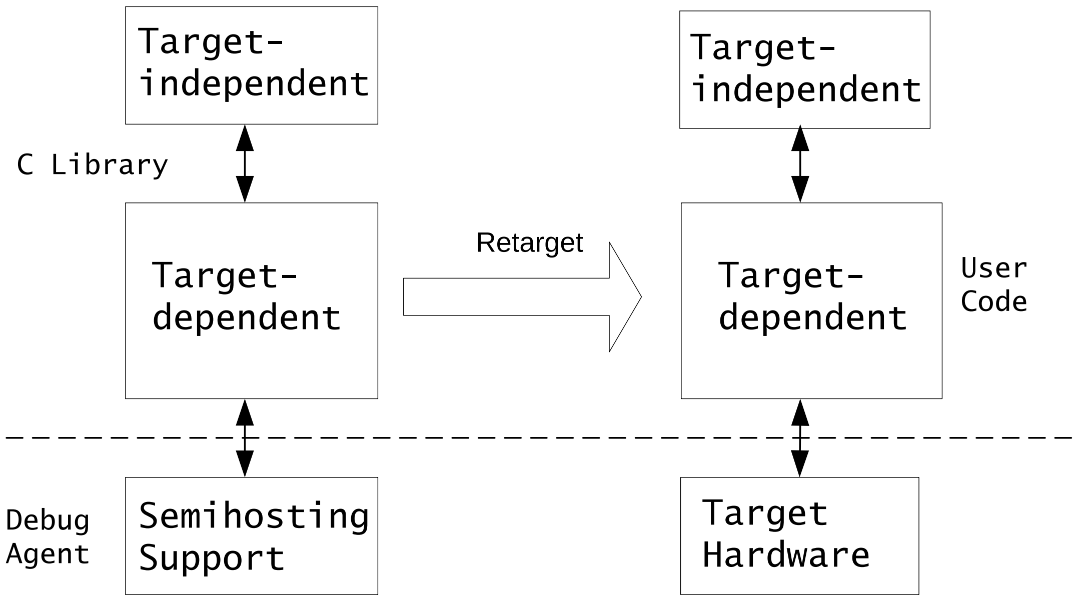


Figure 10-5 Retargeting the C library

For example, you have a peripheral I/O device, such as an LCD screen, and 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 re-implement them with non-semihosting functions.

Related information

Using the libraries in a nonsemihosting environment

Semihosting for AArch32 and AArch64

10.7 Reimplementing C library functions

This provides information for building applications without the Arm standard C library.

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 `--no_scanlib` `armlink` option if you invoke the linker separately.

You must also use the `-fno-builtin` `armclang` option 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.

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).
```

———— **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 using the linker `--startup` option.

Related concepts

10.3 C library structure on page 10-179

Related information

--startup (armlink)

Run-time ABI for the Arm Architecture

C Library ABI for the Arm Architecture

10.8 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/data and 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 Armv6-M and Armv7-M architectures are fixed. This 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

10.9 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:

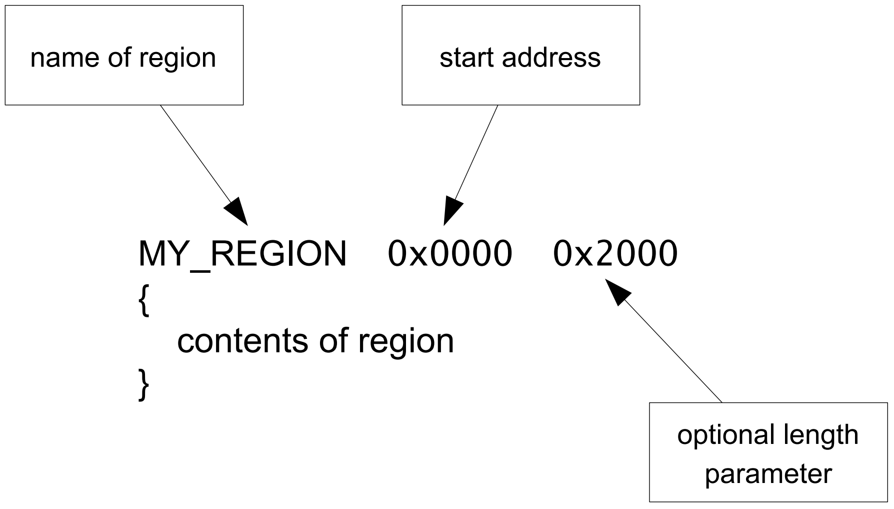


Figure 10-6 Scatter-loading description syntax

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

10.10 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

About placing Arm C and C++ library code

10.11 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 re-implement the `__user_setup_stackheap()` function.

Related concepts

10.12 Run-time memory models on page 10-191

Related information

Tailoring the C library to a new execution environment

Specifying stack and heap using the scatter file

10.12 Run-time memory models

Arm Compiler 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.

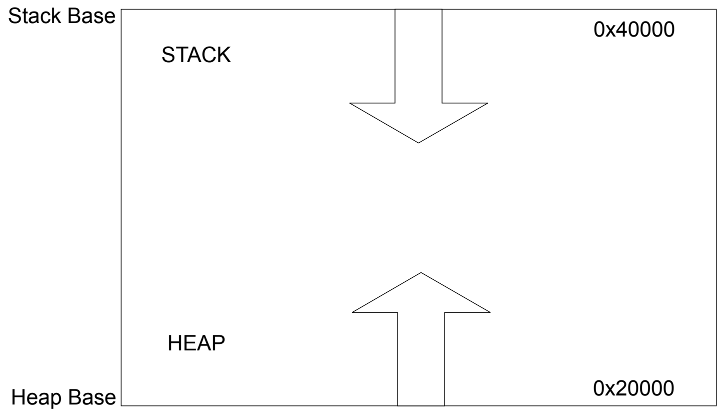


Figure 10-7 One-region model

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.

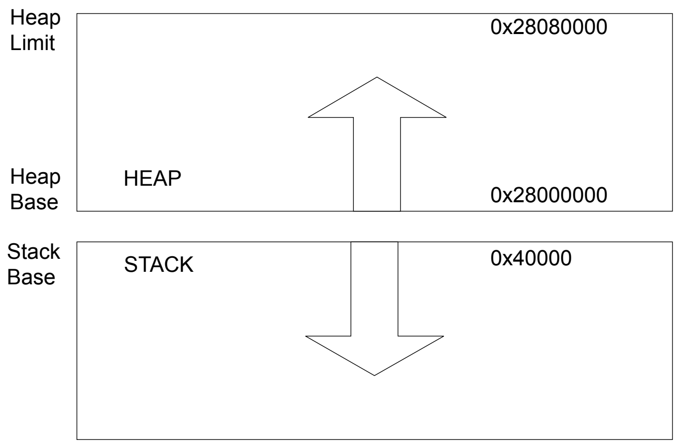


Figure 10-8 Two-region model

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

10.13 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:

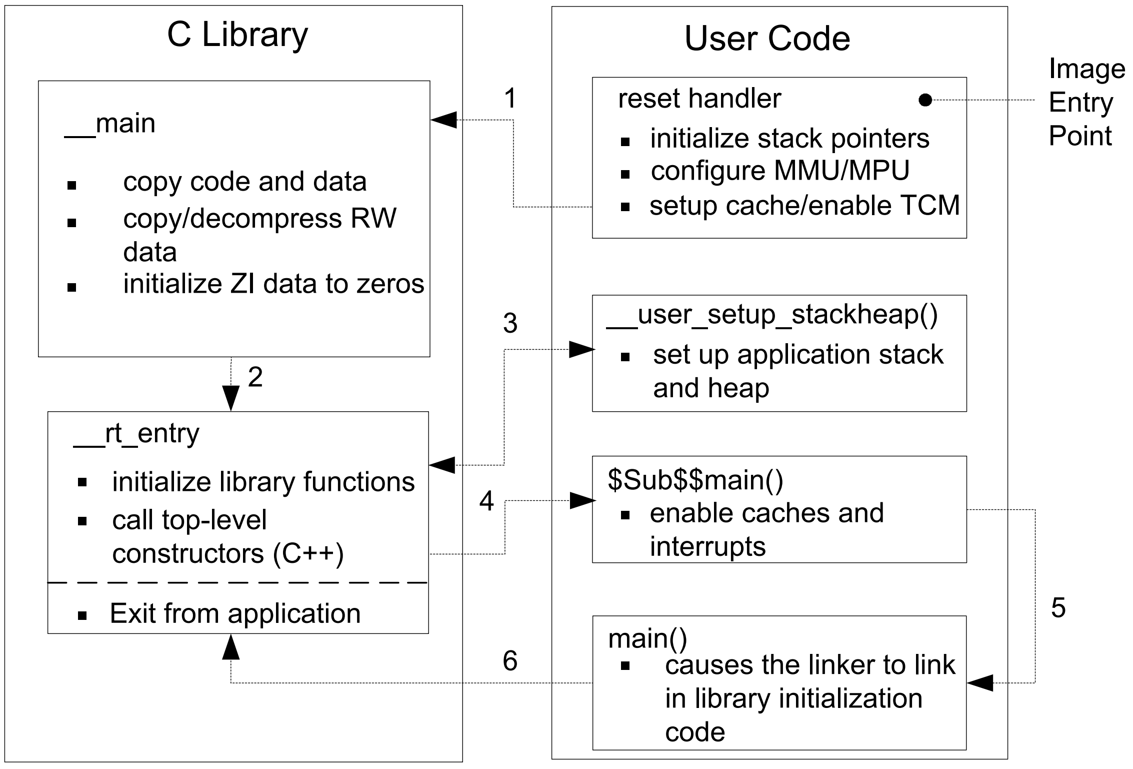

C Library
__main
copy code and data
copy/decompress RW data
initialize ZI data to zeros
2
__rt_entry
initialize library functions
call top-level constructors (C++)
Exit from application
1
3
4
6
User Code
reset handler
initialize stack pointers
configure MMU/MPU
setup cache/enable TCM
Image Entry Point
__user_setup_stackheap()
set up application stack and heap
$Sub$$main()
enable caches and interrupts
5
main()
causes the linker to link in library initialization code

Figure 10-9 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

10.14 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 concepts

10.23 Vector table for ARMv6 and earlier, ARMv7-A and ARMv7-R profiles on page 10-203

10.24 Vector table for M-profile architectures on page 10-204

Related information

Information about scatter files

Scatter-loading images with a simple memory map

10.15 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 Armv6-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
```

10.16 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

10.17 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 Armv6-M and Armv7-M profiles
; ****************************************************************
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

10.18 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

10.19 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 Armv6-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.

————————————

10.20 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 tasks

8.6 Placing functions and data at specific addresses on page 8-132

Related information

__attribute__((section("name"))) variable attribute

10.21 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 tasks

10.22 Building applications for execute-only memory on page 10-202

10.22 Building applications for execute-only memory

Placing code in execute-only memory prevents users from trivially reading that code.

Note

LTO does not honor the `armclang -mexecute-only` option. If you use the `armclang -flto` or `-Omax` options, then the compiler cannot generate execute-only code.

To build an application with code in execute-only memory:

Procedure

1. Compile your C or C++ code using the `-mexecute-only` option.
   Example: `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.

   Example: `armlink --xo-base=0x8000 test.o -o test.axf`
   Results:
   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 concepts

10.21 Execute-only memory on page 10-201

Related information

-mexecute-only (armclang)

--execute_only (armasm)

--xo_base=address (armlink)

AREA directive

10.23 Vector table for ARMv6 and earlier, ARMv7-A and ARMv7-R profiles

The vector table for Armv6 and earlier, Armv7-A and Armv7-R profiles 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

```
                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.

Note

The vector table for Armv6 and earlier architectures supports A32 instructions only. Armv6T2 and later architectures support both T32 instructions and A32 instructions in the vector table. This does not apply to the Armv6-M, Armv7-M, and Armv8-M profiles.

10.24 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 Armv7-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 toolchain sets these bits for you.

————————————

10.25 Vector Table Offset Register

In Armv7-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 minimal alignment is 128 bytes, which allows for 32 exceptions. 16 registers are reserved for system exceptions, and therefore, you can use for 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 (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.

10.26 Integer division-by-zero errors in C code

For targets that do not support hardware division instructions (for example `SDIV` and `UDIV`), you can trap and identify integer division-by-zero errors with the appropriate C library helper functions, `__aeabi_idiv0()` and `__rt_raise()`.

Trapping integer division-by-zero errors with __aeabi_idiv0()

You can trap integer division-by-zero errors with the C library helper function `__aeabi_idiv0()` so that division by zero returns some standard result, for example zero.

Integer division is implemented in code through the C library helper functions `__aeabi_idiv()` and `__aeabi_uidiv()`. Both functions check for division by zero.

When integer division by zero is detected, a branch to `__aeabi_idiv0()` is made. To trap the division by zero, therefore, you only have to place a breakpoint on `__aeabi_idiv0()`.

The library provides two implementations of `__aeabi_idiv0()`. The default one does nothing, so if division by zero is detected, the division function returns zero. However, if you use signal handling, an alternative implementation is selected that calls `__rt_raise(SIGFPE, DIVBYZERO)`.

If you provide your own version of `__aeabi_idiv0()`, then the division functions call this function. The function prototype for `__aeabi_idiv0()` is:

```
int __aeabi_idiv0(void);
```

If `__aeabi_idiv0()` returns a value, that value is used as the quotient returned by the division function.

On entry into `__aeabi_idiv0()`, the link register `LR` contains the address of the instruction *after* the call to the `__aeabi_uidiv()` division routine in your application code.

The offending line in the source code can be identified by looking up the line of C code in the debugger at the address given by `LR`.

If you want to examine parameters and save them for postmortem debugging when trapping `__aeabi_idiv0`, you can use the `$Super$$` and `$Sub$$` mechanism:
1. Prefix `__aeabi_idiv0()` with `$Super$$` to identify the original unpatched function `__aeabi_idiv0()`.
2. Use `__aeabi_idiv0()` prefixed with `$Super$$` to call the original function directly.
3. Prefix `__aeabi_idiv0()` with `$Sub$$` to identify the new function to be called in place of the original version of `__aeabi_idiv0()`.
4. Use `__aeabi_idiv0()` prefixed with `$Sub$$` to add processing before or after the original function `__aeabi_idiv0()`.

The following example shows how to intercept `__aeabi_div0` using the `$Super$$` and `$Sub$$` mechanism.

```
extern void $Super$$__aeabi_idiv0(void);
/* this function is called instead of the original __aeabi_idiv0() */
void $Sub$$__aeabi_idiv0()
{
    // insert code to process a divide by zero
    ...
    // call the original __aeabi_idiv0 function
    $Super$$__aeabi_idiv0();
}
```

Trapping integer division-by-zero errors with __rt_raise()

By default, integer division by zero returns zero. If you want to intercept division by zero, you can re-implement the C library helper function `__rt_raise()`.

The function prototype for `__rt_raise()` is:

```
void __rt_raise(int signal, int type);
```

If you re-implement `__rt_raise()`, then the library automatically provides the signal-handling library version of `__aeabi_idiv0()`, which calls `__rt_raise()`, then that library version of `__aeabi_idiv0()` is included in the final image.

In that case, when a divide-by-zero error occurs, `__aeabi_idiv0()` calls `__rt_raise(SIGFPE, DIVBYZERO)`. Therefore, if you re-implement `__rt_raise()`, you must check `(signal == SIGFPE) && (type == DIVBYZERO)` to determine if division by zero has occurred.

Related information

Use of $Super$$ and $Sub$$ to patch symbol definitions

Chapter 11
Building Secure and Non-secure Images Using Arm®v8-M Security Extensions

Describes how to use the Armv8-M Security Extensions to build a secure image, and how to allow a non-secure image to call a secure image.

It contains the following sections:

- *11.1 Overview of building Secure and Non-secure images* on page 11-209.
- *11.2 Building a Secure image using the Arm®v8-M Security Extensions* on page 11-212.
- *11.3 Building a Non-secure image that can call a Secure image* on page 11-216.
- *11.4 Building a Secure image using a previously generated import library* on page 11-218.

11.1 Overview of building Secure and Non-secure images

Arm Compiler tools allow you to build images that run in the Secure state of the Armv8-M Security Extensions. 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.

————————————

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 -mcmse` command-line option. 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:

- 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 `__ARM_FEATURE_CMSE` predefined macro 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 `-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.

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
```

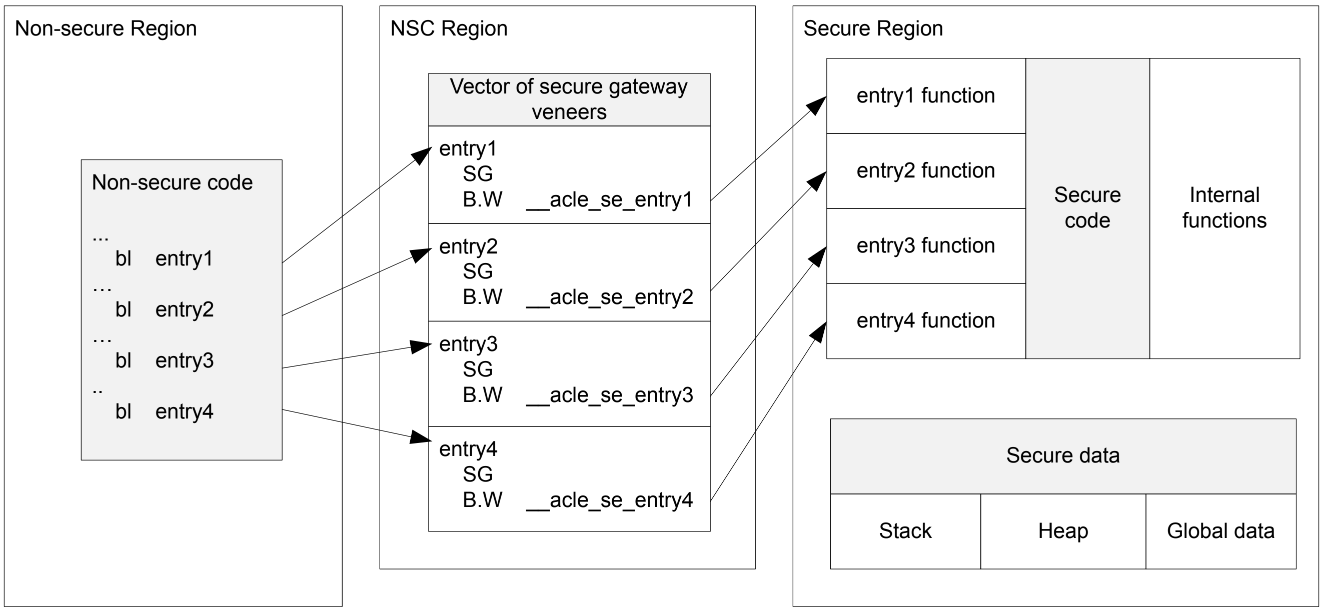
2. The Secure gateway veneer consists of the `SG` instruction and a call to the entry function in the Secure image using the `B` instruction:

```
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 tasks

11.2 Building a Secure image using the Arm®v8-M Security Extensions on page 11-212
11.4 Building a Secure image using a previously generated import library on page 11-218
11.3 Building a Non-secure image that can call a Secure image on page 11-216

Related information

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

11.2 Building a Secure image using the Arm®v8-M Security Extensions

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.

Prerequisites

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.

Procedure

1. Create an interface header file, `myinterface_v1.h`, to specify the C linkage for use by Non-secure code:
   Example:

```
#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:
   Example:

```
#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` `-mcmse` command-line options:
   Example:

```
$ 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:
   Example:

```
$ 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
@ BB#0:
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl func1
    ...
    pop.w {r7, lr}
    ...
    bxns lr
    ...
__acle_se_entry2:
entry2:
    .fnstart
@ BB#0:
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl entry1
    ...
    pop.w {r7, lr}
    bxns lr
    ...
main:
    .fnstart
@ BB#0:
    ...
    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.
   Example:

```
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 --import-cmse-lib-out` command-line option and the scatter file to create the Secure image:
   Example:

```
$ 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:
   Example:

```
$ 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 tasks

11.4 Building a Secure image using a previously generated import library on page 11-218
11.3 Building a Non-secure image that can call a Secure image on page 11-216

Related information

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

11.3 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.

Prerequisites

The following procedure assumes that you have the import library package that is created in *11.2 Building a Secure image using the Arm®v8-M Security Extensions* on page 11-212. 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.
   Example:

```
#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`.
   Example:

```
$ 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.
   Example:

```
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.
   Example:

```
$ armlink nonsecure.o importlib_v1.o -o nonsecure.axf --cpu=8-M.Main --scatter
nonsecure.scat
```

Related tasks

11.2 Building a Secure image using the Arm®v8-M Security Extensions on page 11-212

Related information

Whitepaper - Armv8-M Architecture Technical Overview

-march armclang option

--target armclang option

--cpu armlink option

--scatter armlink option

11.4 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.

Prerequisites

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 *11.2 Building a Secure image using the Arm®v8-M Security Extensions* on page 11-212.

Procedure

1. Create an interface header file, `myinterface_v2.h`, to specify the C linkage for use by Non-secure code:
   Example:

   ```
   #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`:
   Example:

   ```
   #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 -mcmse` command-line options:
   Example:

   ```
   $ 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:
   Example:

   ```
   $ 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
@ BB#0:
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl func1
    pop.w {r7, lr}
    ...
    bxns lr

    ...

__acle_se_entry4:
entry4:
    .fnstart
@ BB#0:
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl entry1
    ...
    pop.w {r7, lr}
    bxns lr

    ...

main:
    .fnstart
@ BB#0:
    ...
    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.
   Example:

```
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` `--import-cmse-lib-out` and `--import-cmse-lib-in` command-line option, together with the preprocessed scatter file to create the Secure image:
   Example:

```
$ 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:
Example:

```
$ 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 tasks

11.2 Building a Secure image using the Arm®v8-M Security Extensions on page 11-212

11.3 Building a Non-secure image that can call a Secure image on page 11-216

Related information

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

Chapter 12
Overview of the Linker

Gives an overview of the Arm linker, `armlink`.

It contains the following sections:

- *12.1 About the linker* on page 12-223.
- *12.2 armlink command-line syntax* on page 12-226.
- *12.3 What the linker does when constructing an executable image* on page 12-227.

12.1 About 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.

This section contains the following subsections:

- *12.1.1 Summary of the linker features* on page 12-223.
- *12.1.2 What the linker can accept as input* on page 12-224.
- *12.1.3 What the linker outputs* on page 12-224.

12.1.1 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 Armv8-M Security Extensions.

———— **Note** ————

Be aware of the following:

- Generated code might be different between two Arm Compiler 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 Reference Guide* describe all the features that Arm Compiler 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* on page Appx-A-262 is operating correctly.

————————————

Related references

Chapter 13 Getting Image Details on page 13-228

Related information

Linker support for creating demand-paged files

Linking Models Supported by armlink

Image Structure and Generation

Linker Optimization Features

Accessing and Managing Symbols with armlink

Scatter-loading Features

BPABI Shared Libraries and Executables

Features of the Base Platform Linking Model

Placement of CMSE veneer sections for a Secure image
Base Platform ABI for the Arm Architecture

12.1.2 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 concepts
17.1 About the Arm® Librarian on page 17-253
Related references
Chapter 11 Building Secure and Non-secure Images Using Arm®v8-M Security Extensions on page 11-208
Related information
--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 (IHI 0044)
ELF for the Arm 64-bit Architecture (AArch64) (IHI 0056)

12.1.3 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 references
Chapter 11 Building Secure and Non-secure Images Using Arm®v8-M Security Extensions on page 11-208
Chapter 15 Overview of the fromelf Image Converter on page 15-238
Related information
Partial linking model
Section placement with the linker

The structure of an Arm ELF image
--import_cmse_lib_out=filename

12.2 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

12.3 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

Chapter 13
Getting Image Details

Describes how to get image details from the Arm linker, `armlink`.

It contains the following sections:

- *13.1 Options for getting information about linker-generated files* on page 13-229.
- *13.2 Identifying the source of some link errors* on page 13-230.
- *13.3 Example of using the --info linker option* on page 13-231.
- *13.4 How to find where a symbol is placed when linking* on page 13-234.

13.1 Options for getting information about linker-generated files

The linker provides options for getting information about the files it generates.

You can use 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.

Related tasks

13.2 Identifying the source of some link errors on page 13-230

Related references

13.3 Example of using the --info linker option on page 13-231

Related information

--info=topic[,topic,…]

Section alignment with the linker

Optimization with RW data compression

--list=filename (armlink)

--map, --no_map (armlink)

--show_cmdline (armlink)

--symbols, --no_symbols (armlink)

--verbose (armlink)

--xref, --no_xref (armlink)

--xrefdbg, --no_xrefdbg (armlink)

13.2 Identifying the source of some link errors

The linker provides options to help you identify the source of some link errors.

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 references

13.1 Options for getting information about linker-generated files on page 13-229

Related information

--info=topic[,topic,...] (armlink)

--verbose (armlink)

13.3 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 references

13.1 Options for getting information about linker-generated files on page 13-229

Related information

--info=topic[,topic,...] (armlink)

13.4 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.

You can do this 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 tasks

16.6 Using fromelf to find where a symbol is placed in an executable ELF image on page 16-250

Related information

--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

Chapter 14
SysV Dynamic Linking

Arm Compiler 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.

———————————

It contains the following sections:

- *14.1 Build a SysV shared object* on page 14-236.
- *14.2 Build a SysV executable* on page 14-237.

14.1 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`.

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
...
```

14.2 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.

Build the image and then run `fromelf` to examine the contents.

Prerequisites

Build the `lib.o` shared library as described in *14.1 Build a SysV shared object* on page 14-236.

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.
```

Chapter 15
Overview of the fromelf Image Converter

Gives an overview of the `fromelf` image converter provided with Arm Compiler.

It contains the following sections:

- *15.1 About the fromelf image converter* on page 15-239.
- *15.2 fromelf execution modes* on page 15-240.
- *15.3 Getting help on the fromelf command* on page 15-241.
- *15.4 fromelf command-line syntax* on page 15-242.

15.1 About 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 Armv8.4-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 Reference Guide* describe all the features that Arm Compiler 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* on page Appx-A-262 is operating correctly.

Related concepts

16.3 Options to protect code in image files with fromelf on page 16-246

16.4 Options to protect code in object files with fromelf on page 16-247

Related references

15.2 fromelf execution modes on page 15-240

15.4 fromelf command-line syntax on page 15-242

Related information

fromelf Command-line Options

15.2 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)

15.3 Getting help on the fromelf 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 files.

To display the help information, enter:

```
fromelf --help
```

Related references

15.4 fromelf command-line syntax on page 15-242

Related information

--help (fromelf)

15.4 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)

Chapter 16
Using fromelf

Describes how to use the `fromelf` image converter provided with Arm Compiler.

It contains the following sections:

- *16.1 General considerations when using fromelf* on page 16-244.
- *16.2 Examples of processing ELF files in an archive* on page 16-245.
- *16.3 Options to protect code in image files with fromelf* on page 16-246.
- *16.4 Options to protect code in object files with fromelf* on page 16-247.
- *16.5 Option to print specific details of ELF files* on page 16-249.
- *16.6 Using fromelf to find where a symbol is placed in an executable ELF image* on page 16-250.

16.1 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)

16.2 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/
```

This 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/
```

This 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*)
```

Related information

--disassemble (fromelf)

--elf (fromelf)

input_file (fromelf)

--output=destination (fromelf)

--strip=option[,option,…] (fromelf)

16.3 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 `fromelf --text` output. 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 concepts

16.4 Options to protect code in object files with fromelf on page 16-247

Related references

15.4 fromelf command-line syntax on page 15-242

Related information

--elf (fromelf)

--output=destination (fromelf)

--privacy (fromelf)

--strip=option[,option,…] (fromelf)

16.4 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 concepts

16.3 Options to protect code in image files with fromelf on page 16-246

Related references

15.4 fromelf command-line syntax on page 15-242

Related information

--elf (fromelf)

--output=destination (fromelf)

--privacy (fromelf)

--strip=option[,option,…] (fromelf)

16.5 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 references

15.4 fromelf command-line syntax on page 15-242

Related information

--emit=option[,option,…] (fromelf)

--text (fromelf)

16.6 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.

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)

Chapter 17
Overview of the Arm® Librarian

Gives an overview of the Arm Librarian, `armar`, provided with Arm Compiler.

It contains the following sections:

- *17.1 About the Arm® Librarian* on page 17-253.
- *17.2 Considerations when working with library files* on page 17-254.
- *17.3 armar command-line syntax* on page 17-255.
- *17.4 Option to get help on the armar command* on page 17-256.

17.1 About 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.

Related information

--debug_symbols (armar)

--library=name (armlink)

--libpath=pathlist (armlink)

--library_type=lib (armlink)

--userlibpath=pathlist (armlink)

17.2 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 --remove` option 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)

17.3 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)

17.4 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
```

Chapter 18
Overview of the armasm Legacy Assembler

Gives an overview of the `armasm` legacy assembler provided with Arm Compiler toolchain.

It contains the following sections:

- *18.1 Key features of the armasm assembler* on page 18-258.
- *18.2 How the assembler works* on page 18-259.

18.1 Key features of the armasm assembler

The `armasm` assembler supports instructions, directives, and user-defined macros.

It supports:

- *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 Armv8.2-A and Armv8.3-A .

Note

`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 Reference Guide*.
- AArch64 Crypto instructions (for SHA512, SHA3, SM3, SM4). See `+crypto` in the *-mcpu* command-line option in the *Arm Compiler Reference Guide*.
- AArch64 *Scalable Vector Extension* (SVE) instructions. See `+sve` in the *-mcpu* command-line option in the *Arm Compiler Reference Guide*.

Related concepts

18.2 How the assembler works on page 18-259

Related information

About the Unified Assembler Language

Use of macros

armasm Directives Reference

--cpu=name (armasm)

-mcpu

Arm Compiler Instruction Set Reference Guide

18.2 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

Appendix A
Supporting reference information

The various features in Arm Compiler might have different levels of support, ranging from fully supported product features to community features.

It contains the following sections:

- *A.1 Support level definitions* on page Appx-A-262.
- *A.2 Standards compliance in Arm® Compiler* on page Appx-A-266.
- *A.3 Compliance with the ABI for the Arm® Architecture (Base Standard)* on page Appx-A-267.
- *A.4 GCC compatibility provided by Arm® Compiler 6* on page Appx-A-269.
- *A.5 Locale support in Arm® Compiler* on page Appx-A-270.
- *A.6 Toolchain environment variables* on page Appx-A-271.
- *A.7 Clang and LLVM documentation* on page Appx-A-273.
- *A.8 Further reading* on page Appx-A-274.

A.1 Support level definitions

This describes the levels of support for various Arm Compiler 6 features.

Arm Compiler 6 is built on Clang and LLVM technology. Therefore, it has more functionality than the set of product features described in the documentation. The following definitions clarify the levels of support and guarantees on functionality that are expected from these features.

Arm welcomes feedback regarding the use of all Arm Compiler 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*.

Identification in the documentation

All features that are documented in the Arm Compiler 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.

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 indicated 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 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.

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 indicated 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.

Community features

Arm Compiler 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 that are not listed in the documentation. These additional features are known as community features. For information on these community features, see the *documentation for the Clang/LLVM project*.

Where community features are referenced in the documentation, they are indicated 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 will 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 will 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 6 toolchain:

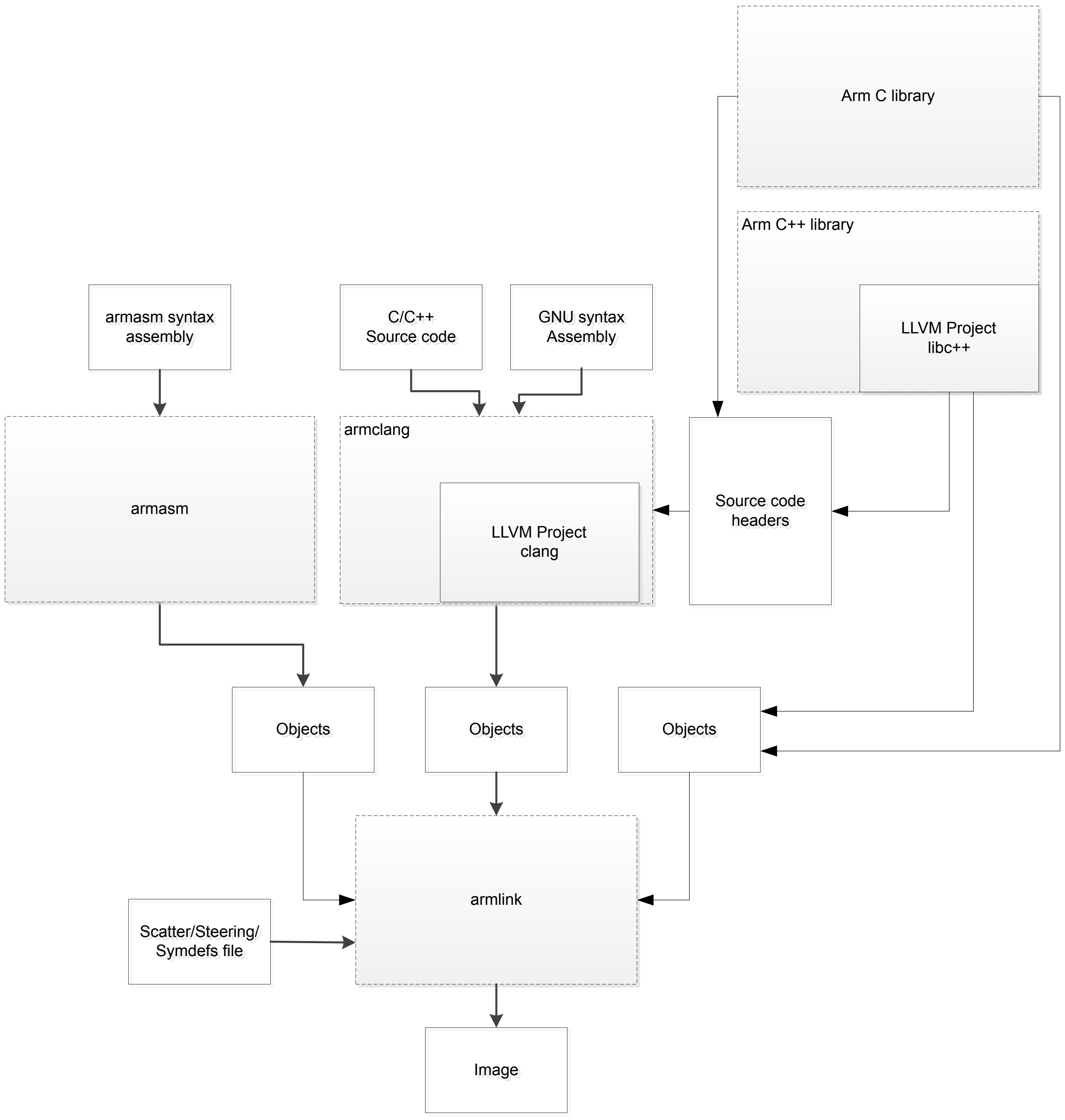


Figure A-1 Integration boundaries in Arm Compiler 6.

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 6. See *Application Binary Interface (ABI) for the Arm® Architecture*. 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. Arm does not make any guarantee regarding the testing or maintenance of deprecated features. Therefore, Arm does not recommend using a feature after it is deprecated.

For information on replacing deprecated features with supported features, refer to the Arm Compiler documentation and Release Notes.

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 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* on page Appx-A-262.

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.
- 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.
- Compilation for targets that implement architectures older than Armv7 or Armv6-M 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.
- C++ complex numbers are supported with **`float`** and **`double`** types, but not **`long double`** type because of limitations in the current Arm C library.

A.2 Standards compliance in Arm® Compiler

Arm Compiler 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 3 or DWARF 2 for backwards compatibility with legacy and third-party tools.

The linker and the `fromelf` utility can consume ELF format inputs containing DWARF 4, DWARF 3, and DWARF 2 format debug tables.

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, and C++14 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 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 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++ Status

A.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). 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 6 implements the LP64 data model for AArch64 state.

AAPCS
Procedure Call Standard for the Arm® 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.

BPABI
Base Platform ABI for the Arm® Architecture. Governs the format and content of executable and shared object files generated by static linkers. Supports platform-specific executable files using post linking. Provides a base standard for deriving a platform ABI.

CLIBABI
C Library ABI for the Arm® Architecture. Defines an ABI to the C library.

CPPABI64
C++ ABI for the Arm® Architecture. This specification builds on the generic C++ ABI (originally developed for IA-64) 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.

A.4 GCC compatibility provided by Arm® Compiler 6

The compiler in Arm Compiler 6 is based on Clang and LLVM technology. As such, it provides a high degree of compatibility with GCC.

Arm Compiler 6 can build most of the C code that is written to be built with GCC. However, Arm Compiler is not 100% source compatible in all cases. Specifically, Arm Compiler does not aim to be bug-compatible with GCC. That is, Arm Compiler does not replicate GCC bugs.

A.5 Locale support in Arm® Compiler

Summarizes the locales supported by Arm Compiler.

Arm Compiler provides full support only for the English locale.

Arm Compiler 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.

————————————

A.6 Toolchain environment variables

Except for `ARMLMD_LICENSE_FILE`, Arm Compiler does not require any other 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 *Environment Variables Affecting GCC* on the *GCC web site*.

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 ARM_TOOL_VARIANT=ult
```

Table A-1 Environment variables used by the toolchain

Environment variable	Setting
`ARM_PRODUCT_PATH`	Required only if you have an Arm Development Studio or Arm DS-5 Development Studio toolkit license and you are running the Arm Compiler tools outside of the that environment. Use this environment variable to specify the location of the `sw/mappings` directory within an Arm Compiler, Arm Development Studio, or DS-5 Development Studio installation.
`ARM_TOOL_VARIANT`	Required only if you have an Arm Development Studio or DS-5 Development Studio toolkit license and you are running the Arm Compiler tools outside of that environment. If you have an ultimate license, set this environment variable to `ult` to enable the Ultimate features. See *Product and toolkit configuration* for more information.
`ARM_PRODUCT_DEF`	Required only if you have an Arm Development Studio toolkit license and you are running the Arm Compiler tools outside of the Arm Development Studio environment. Use this environment variable to specify the location of the product definition file. See *Product and toolkit configuration* for more information.
`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.

Table A-1 Environment variables used by the toolchain (continued)

Environment variable	Setting
`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`*.
`ARMLMD_LICENSE_FILE`	This environment variable must be set, and specifies the location of your Arm license file. See the *Arm® DS-5 License Management Guide* for information on this environment variable. ——— **Note** ——— On Windows, the length of `ARMLMD_LICENSE_FILE` must not exceed 260 characters. ———————
`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.
`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.

Related information

Product and toolkit configuration

A.7 Clang and LLVM documentation

Arm Compiler is based on Clang and LLVM compiler technology.

The Arm Compiler documentation describes features that are specific to, and supported by, Arm Compiler. Any features specific to Arm Compiler 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* on page Appx-A-262 is operating correctly.

The *Clang Compiler User's Manual*, available from the LLVM Compiler Infrastructure Project web site *`http://clang.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 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.

A.8 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. See *Arm® Infocenter* for current errata sheets and addenda, and the Arm Frequently Asked Questions (FAQs).

For full information about the base standard, software interfaces, and standards supported by Arm, see *Application Binary Interface (ABI) for the Arm® Architecture*.

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*.

Other publications

This Arm Compiler 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 14882:2014, 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 *The DWARF Debugging Standard web site* for the latest information about the *Debug With Arbitrary Record Format* (DWARF) debug table standards and ELF specifications.

Appendix B
Arm® Compiler User Guide Changes

Describes the technical changes that have been made to the Arm Compiler User Guide.

It contains the following section:

- *B.1 Changes for the Arm Compiler User Guide* on page Appx-B-277.

B.1 Changes for the Arm Compiler User Guide

Changes that have been made to the Arm Compiler User Guide are listed with the latest version first.

Table B-1 Changes between 6.15 and 6.14

Changes	Topics affected
Added chapters about the Scalable Vector Extension (SVE) compiler.	• *Chapter 2 Getting Started with the SVE features in Arm® Compiler* on page 2-30. • *Chapter 7 SVE Coding Considerations with Arm® Compiler* on page 7-110.
Added note about Arm Compiler and undefined behavior.	• *3.3 Selecting source language options* on page 3-44. • *A.2 Standards compliance in Arm® Compiler* on page Appx-A-266.
Added note about not specifying both the architecture (-march) and the processor (-mcpu).	• *3.1 Mandatory armclang options* on page 3-39. • *3.10 Selecting floating-point options* on page 3-61.
Added details about the SVE and SVE2 intrinsics support.	• *7.2 Using SVE and SVE2 intrinsics directly in your C code* on page 7-116.
Reworded the note about dynamic linking not being supported for Cortex-M processors.	• *Chapter 14 SysV Dynamic Linking* on page 14-235.
Added note clarifying that Arm Compiler 6 is not based on the same revision as any specific release of the open source version of LLVM and Clang, and may contain Arm-specific changes which are not included in open source versions.	• *A.7 Clang and LLVM documentation* on page Appx-A-273.
Updated text and examples to clarify correct naming of sections when using #pragma clang section.	• *4.9 Scatter file section or object placement with Link-Time Optimization* on page 4-91.
Added note that all eXecute In Place (XIP) code must be stored in root regions.	• *8.4 Root region* on page 8-127. • *10.10 Root regions* on page 10-189.
Improved explanation of when to use the volatile keyword to prevent unwanted removal of inline assembler code when building optimized output.	• *B.1 Changes for the Arm Compiler User Guide* on page Appx-B-277. • *6.2 Writing inline assembly code* on page 6-105.
Added details of the new -Omin compiler option which minimizes code size.	• *3.4 Selecting optimization options* on page 3-48. • *4.6 Optimizing for code size or performance* on page 4-83.
Removed outdated note about using __ARM_use_no_argv with -O0 optimization level in Arm Compiler 6. The -O0 option now supports argv/argc optimization.	• *3.4 Selecting optimization options* on page 3-48.
Progressive terminology commitment added to Proprietary notices section (all documents).	• Proprietary notices