ARM® Compiler

Version 6.3

Software Development Guide

ARM

Copyright © 2014, 2015 ARM. All rights reserved.
ARM DUI0773D

ARM® Compiler

Software Development Guide

Copyright © 2014, 2015 ARM. All rights reserved.

Release Information

Document History

Issue	Date	Confidentiality	Change
A	14 March 2014	Non-Confidential	ARM Compiler v6.00 Release
B	15 December 2014	Non-Confidential	ARM Compiler v6.01 Release
C	30 June 2015	Non-Confidential	ARM Compiler v6.02 Release
D	18 November 2015	Non-Confidential	ARM Compiler v6.3 Release

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of ARM. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

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

This document may include technical inaccuracies or typographical errors.

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

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

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow ARM's trademark usage guidelines at *http://www.arm.com/about/trademark-usage-guidelines.php*

Copyright © [2014, 2015], ARM Limited or its affiliates. All rights reserved.

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

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

http://www.arm.com

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Contents
ARM® Compiler Software Development Guide

Preface

About this book 9

Chapter 1 Introducing the Toolchain

1.1 Toolchain overview 1-12
1.2 Support level definitions 1-13
1.3 LLVM component versions and language compatibility 1-16
1.4 Common ARM Compiler toolchain options 1-18
1.5 "Hello world" example 1-21
1.6 Passing options from the compiler to the linker 1-22

Chapter 2 Diagnostics

2.1 Understanding diagnostics 2-24
2.2 Options for controlling diagnostics with armclang 2-26
2.3 Options for controlling diagnostics with the other tools 2-27

Chapter 3 Compiling C and C++ Code

3.1 Specifying a target architecture, processor, and instruction set 3-29
3.2 Using C++ libraries 3-32
3.3 Using inline assembly code 3-33
3.4 Using intrinsics 3-34
3.5 Preventing the use of floating-point instructions and registers 3-35
3.6 Bare-metal Position Independent Executables 3-36

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Chapter 4 **Assembling Assembly Code**
4.1 Assembling ARM and GNU syntax assembly code 4-39
4.2 Preprocessing assembly code 4-41

Chapter 5 **Linking Object Files to Produce an Executable**
5.1 Linking object files to produce an executable 5-43

Chapter 6 **Optimization**
6.1 Optimizing for code size or performance 6-45
6.2 Optimizing across modules with link time optimization 6-46
6.3 How optimization affects the debug experience 6-50

Chapter 7 **Coding Considerations**
7.1 Optimization of loop termination in C code 7-52
7.2 Loop unrolling in C code 7-54
7.3 Compiler optimization and the volatile keyword 7-56
7.4 Stack use in C and C++ 7-58
7.5 Methods of minimizing function parameter passing overhead 7-60
7.6 Inline functions 7-61
7.7 Integer division-by-zero errors in C code 7-62
7.8 Infinite Loops 7-64

Chapter 8 **Using the ARMv8-M Security Extensions [BETA]**
8.1 Overview of ARMv8-M Security Extensions [BETA] 8-66
8.2 Creating a custom import library [BETA] 8-74
8.3 Generating a secure gateway veneer for a secure image [BETA] 8-75
8.4 Using an import library when building a non-secure image [BETA] 8-78

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

List of Figures
ARM® Compiler Software Development Guide

Figure 1-1	*Compiler toolchain*	*1-12*
Figure 1-2	*Integration boundaries in ARM Compiler 6.*	*1-14*
Figure 6-1	*Link time optimization*	*6-46*

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

List of Tables
ARM® Compiler Software Development Guide

Table 1-1	LLVM component versions	1-16
Table 1-2	Language support levels	1-16
Table 1-3	armclang common options	1-18
Table 1-4	armlink common options	1-19
Table 1-5	armar common options	1-19
Table 1-6	fromelf common options	1-20
Table 1-7	armasm common options	1-20
Table 1-8	armclang linker control options	1-22
Table 3-1	Compiling for different combinations of architecture, processor, and instruction set	3-30
Table 7-1	C code for incrementing and decrementing loops	7-52
Table 7-2	C disassembly for incrementing and decrementing loops	7-52
Table 7-3	C code for rolled and unrolled bit-counting loops	7-54
Table 7-4	Disassembly for rolled and unrolled bit-counting loops	7-55
Table 7-5	C code for nonvolatile and volatile buffer loops	7-56
Table 7-6	Disassembly for nonvolatile and volatile buffer loop	7-57

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Preface

This preface introduces the *ARM® Compiler Software Development Guide*.

It contains the following:

- *About this book* on page 9.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

About this book

The ARM® Compiler Software Development Guide provides tutorials and examples to develop code for various ARM architecture-based processors.

Using this book

This book is organized into the following chapters:

Chapter 1 Introducing the Toolchain
Provides an overview of the ARM compilation tools, and shows how to compile a simple code example.

Chapter 2 Diagnostics
Describes the format of compiler toolchain diagnostic messages and how to control the diagnostic output.

Chapter 3 Compiling C and C++ Code
Describes how to compile C and C++ code with `armclang`.

Chapter 4 Assembling Assembly Code
Describes how to assemble assembly source code with `armclang` and `armasm`.

Chapter 5 Linking Object Files to Produce an Executable
Describes how to link object files to produce an executable image with `armlink`.

Chapter 6 Optimization
Describes how to use `armclang` to optimize for either code size or performance, and the impact of the optimization level on the debug experience.

Chapter 7 Coding Considerations
Describes how you can use programming practices and techniques to increase the portability, efficiency and robustness of your C and C++ source code.

Chapter 8 Using the ARMv8-M Security Extensions [BETA]
Describes how to use the ARMv8-M Security Extensions in your applications.

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` (with "mono" underlined)
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.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

`<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 Software Development Guide*.
- The number ARM DUI0773D.
- 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 Information Center*.
- *ARM Technical Support Knowledge Articles*.
- *Support and Maintenance*.
- *ARM Glossary*.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Chapter 1
Introducing the Toolchain

Provides an overview of the ARM compilation tools, and shows how to compile a simple code example.

It contains the following sections:

- *1.1 Toolchain overview* on page 1-12.
- *1.2 Support level definitions* on page 1-13.
- *1.3 LLVM component versions and language compatibility* on page 1-16.
- *1.4 Common ARM Compiler toolchain options* on page 1-18.
- *1.5 "Hello world" example* on page 1-21.
- *1.6 Passing options from the compiler to the linker* on page 1-22.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

1.1 Toolchain overview

The ARM® Compiler 6 compilation tools allow you to build executable images, partially linked object files, and shared object files, and to convert images to different formats.

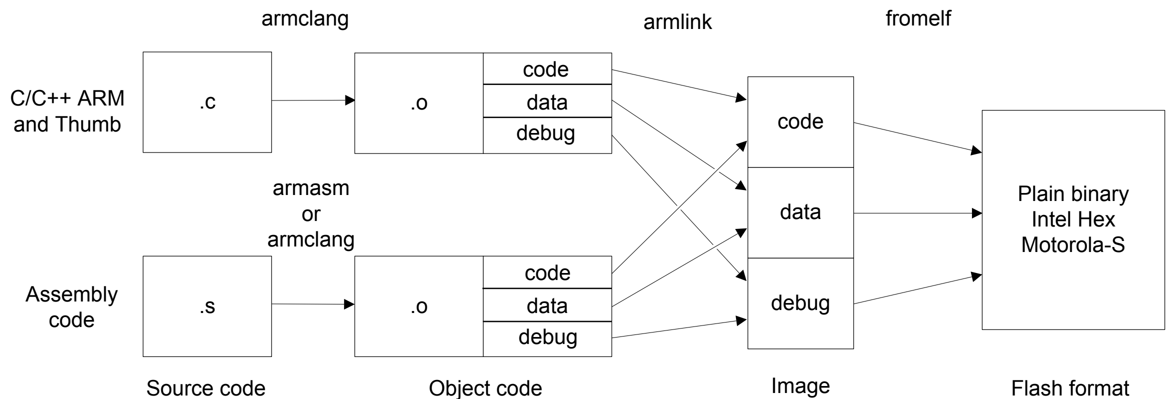


Figure 1-1 Compiler toolchain

The ARM Compiler toolchain comprises the following tools:

armclang
The `armclang` compiler and assembler. This compiles C and C++ code, and assembles A32, A64, and T32 GNU syntax assembly code.

armasm
The legacy assembler. This assembles A32, A64, and T32 assembly code, using ARM syntax.

Only use `armasm` for legacy ARM syntax assembly code. Use the `armclang` assembler and GNU syntax for all new assembly files.

armlink
The linker. This 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 librarian. This enables sets of ELF object files to be collected together and maintained in archives or libraries. 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 for further application development.

fromelf
The image conversion utility. This can also generate textual information about the input image, such as its disassembly and its code and data size.

Related tasks

1.5 "Hello world" example on page 1-21.

Related references

1.4 Common ARM Compiler toolchain options on page 1-18.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

1.2 Support level definitions

Describes the levels of support for various ARM Compiler features.

ARM Compiler 6 is built on Clang and LLVM technology and as such, 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 is expected from these features.

ARM welcomes feedback regarding the use of all ARM Compiler 6 features, and endeavors to support users to a level that is appropriate for that feature. You can contact support at *http://www.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 endeavors 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*.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

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 are going to remain functional across update releases, although changes are expected to be unlikely.

Some community features might become product features in the future, but ARM provides no roadmap for this. ARM is interested in understanding your use of these features, and welcomes feedback on them. ARM supports customers using these features on a best-effort basis, unless the features are unsupported. ARM accepts defect reports on these features, but does not guarantee that these issues are going to be fixed in future releases.

Guidance on use of community features

There are several factors to consider when assessing the likelihood of a community feature being functional:

- The following figure shows the structure of the ARM Compiler 6 toolchain:

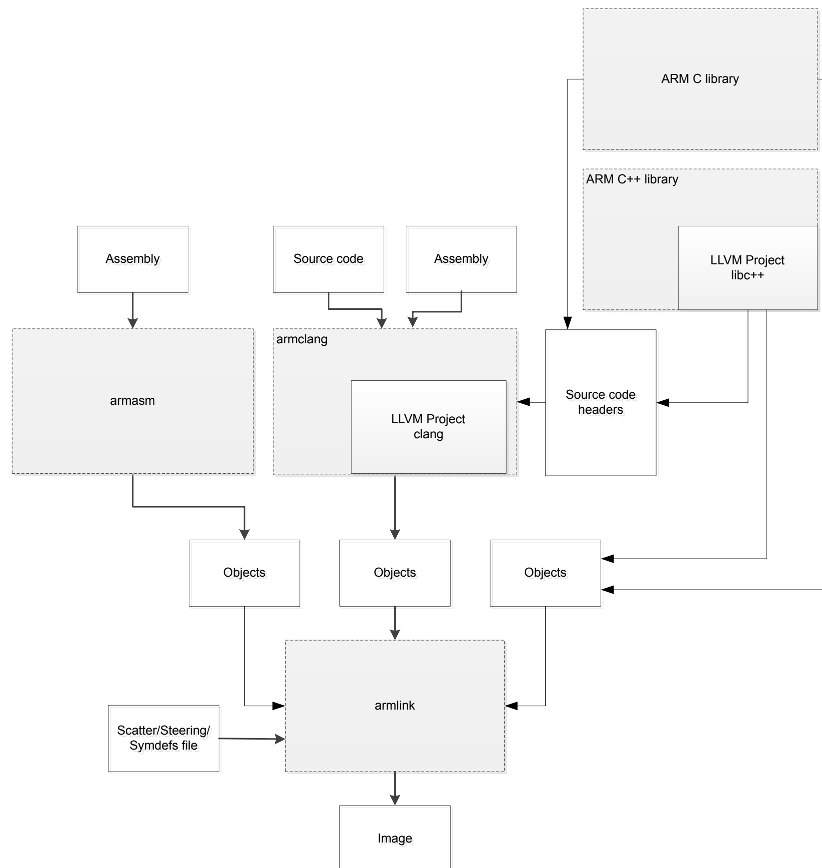


Figure 1-2 Integration boundaries in ARM Compiler 6.

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

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

standards supported by ARM Compiler 6. See *Application Binary Interface (ABI) for the ARM® Architecture*.
- 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.

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 1-13.

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.
- The ARM Compiler 6 libc++ libraries do not support the Atomic operations library <atomic> and the Thread support library <thread>.
- Use of C11 library features is unsupported.
- Any community feature that exclusively pertains to non-ARM architectures is not supported by ARM Compiler 6.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

1.3 LLVM component versions and language compatibility

`armclang` is based on LLVM components and provides different levels of support for different source language standards.

Note

This topic includes descriptions of *[COMMUNITY]* on page 1-13 features.

Base LLVM components

ARM Compiler 6 is based on the following LLVM components:

Table 1-1 LLVM component versions

Component	Version	More information
Clang	3.7	*http://clang.llvm.org*

Language support levels

ARM Compiler 6 in conjunction with libc++ provides varying levels of support for different source language standards:

Table 1-2 Language support levels

Language standard	Support level
C90	Supported.
C99	Supported, with the exception of complex numbers.
[COMMUNITY] C11	The base Clang component provides C11 language functionality. However, ARM has performed no independent testing of these features so they are a community feature. Use of C11 library features is unsupported. Note that C11 is the default language standard for C code. However, usage of the new C11 language features is a community feature. Use the `-std` option to restrict the language standard if required. Use the `-Wc11-extensions` option to warn about any use of C11-specific features.
C++98	Supported, including the use of C++ exceptions. Support for `-fno-exceptions` is limited. See *Standard C++ library implementation definition* in the *ARM C and C++ Libraries and Floating-Point Support User Guide* for more information about support for exceptions.
C++11	Supported, with the following exceptions: • The Atomic operations library is not supported. • The Thread support library is not supported. C++11 can only be used for creating single-threaded operations. See *Standard C++ library implementation definition* in the *ARM C and C++ Libraries and Floating-Point Support User Guide* for more information.
[COMMUNITY] C++14	The base Clang and libc++ components provide C++14 language functionality. However, ARM has performed no independent testing of these features so they are a community feature.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

C++ library variants

ARM Compiler 6 provides two different C++ library variants:

- libc++, based on the LLVM C++ Standard Library. This is the default library.
- Rogue Wave Standard C++ Library.

——— **Note** ———

The Rogue Wave C++ libraries are deprecated.

———————————

Additional information

See the *armclang 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

Related information

armclang Reference Guide.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

1.4 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 *armclang Reference Guide* for more information about `armclang` command-line options.

Common `armclang` options include the following:

Table 1-3 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 `-mcpu=armv8-a` or `-mcpu=armv7-a`.
`-mcpu=`*`name`*	Generates code for the specified processor, for example `-mcpu=cortex-a53`, `-mcpu=cortex-a57`, or `-mcpu=cortex-a15`.
`-marm`	Requests that the compiler targets the A32 instruction set, `--target=arm-arm-none-eabi -march=armv7-a -marm` for example. The `-marm` option is not valid with AArch64 targets. The compiler ignores the `-marm` option and generates a warning with AArch64 targets.
`-mthumb`	Requests that the compiler targets the T32 instruction set, `--target=arm-arm-none-eabi -march=armv8-a -mthumb` for example. The `-mthumb` option is not valid with AArch64 targets. The compiler ignores the `-mthumb` option and generates a warning with AArch64 targets.
`-g`	Generates DWARF debug tables.
`-E`	Executes only the preprocessor step.
`-I`	Adds the specified directories to the list of places that are searched to find included files.
`-o`	Specifies the name of the output file.
`-O`*`num`*	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 generated by the compiler.
`-###`	Displays diagnostic output showing the options that would be used to invoke the compiler and linker. Neither the compilation nor the link steps are performed.

armlink common options

See the *armlink User Guide* for more information about `armlink` command-line options.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Common armlink options include the following:

Table 1-4 armlink common options

Option	Description
--cpu=*name*	Sets the target processor.
--fpu=*name*	Selects the target floating-point unit (FPU) architecture.
--ro_base	Sets the load and execution addresses of the region containing the RO output section to a specified address.
--rw_base	Sets the execution address of the region containing the RW output section to a specified address.
--scatter	Creates an image memory map using the scatter-loading description contained in the specified file.
--split	Splits the default load region containing the RO and RW output sections, into separate regions.
--entry	Specifies the unique initial entry point of the image.
--info	Displays information about linker operation, for example --info=exceptions displays information about exception table generation and optimization.
--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 used in the link step, and their values.

armar common options

See the *armar User Guide* for more information about armar command-line options.

Common armar options include the following:

Table 1-5 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 *fromelf User Guide* for more information about fromelf command-line options.

Common fromelf options include the following:

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Table 1-6 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 *armasm User Guide* for more information about `armasm` command-line options.

Note

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

Common `armasm` options include the following:

Table 1-7 armasm common options

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

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

1.5 "Hello world" example

This example shows how to build a simple C program `hello_world.c` with `armclang` and `armlink`.

Procedure

1. Create a C file `hello_world.c` with the following content:

```
#include <stdio.h>

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

2. Compile the C file `hello_world.c` with the following command:

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

   The `-c` option tells the compiler to perform the compilation step only. The `-march=armv8-a` option tells the compiler to target the ARMv8-A architecture, and `--target=aarch64-arm-none-eabi` targets AArch64 state.

   The compiler creates an object file `hello_world.o`

3. Link the file:

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

   The `-o` option tells the linker to name the output image `hello_world.axf`, rather than using the default image name `__image.axf`.

4. Use a DWARF 4 compatible debugger to load and run the image.

   The compiler produces debug information that is compatible with the DWARF 4 standard.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

1.6 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 1-8 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.
`-rdynamic`	`--export-dynamic`	If an executable has dynamic symbols, export all externally visible symbols rather than only referenced symbols.
`-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
```

Use a file comparison tool, such as the UNIX `diff` tool, to compare the files `hello_DEFAULT.txt` and `hello_SPLIT.txt`.

 Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Chapter 2
Diagnostics

Describes the format of compiler toolchain diagnostic messages and how to control the diagnostic output.

It contains the following sections:

- *2.1 Understanding diagnostics* on page 2-24.
- *2.2 Options for controlling diagnostics with armclang* on page 2-26.
- *2.3 Options for controlling diagnostics with the other tools* on page 2-27.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

2.1 Understanding diagnostics

All the tools in the ARM Compiler 6 toolchain produce detailed diagnostic messages, and let you control how much or how little information is output.

The format of diagnostic messages and the mechanisms for controlling diagnostic output are different for `armclang` than for the other tools in the toolchain.

Message format for armclang

`armclang` produces messages in the following format:

```
file:line:col: type: message
```

where:

file
: The filename that generated the message.

line
: The line number that generated the message.

col
: The column number that generated the message.

type
: The type of the message, for example error or warning.

message
: The message text.

For example:

```
hello.c:7:3: error: use of undeclared identifier 'i'
i++;
^
1 error 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
```

Where:

type
: is one of:

  `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
: indicates the tool that generated the message, one of:

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

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

id
: a unique numeric message identifier.

suffix
: indicates the type of message, one of:
  - E - Error
  - W - Warning
  - R - Remark

message_text
: the text of the message.

For example:

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

Related concepts

2.2 Options for controlling diagnostics with armclang on page 2-26.
2.3 Options for controlling diagnostics with the other tools on page 2-27.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

2.2 Options for controlling diagnostics with armclang

A number of options control the output of diagnostics with 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:

`-Werror`
: Turn warnings into errors.

`-Werror=foo`
: Turn warning `foo` into an error.

`-Wno-error=foo`
: Leave warning `foo` as a warning even if `-Werror` is specified.

`-Wfoo`
: Enable warning `foo`.

`-Wno-foo`
: Suppress warning `foo`.

`-w`
: Suppress all warnings.

`-Weverything`
: Enable all warnings.

Where a message can be suppressed, the compiler provides the appropriate suppression flag in the diagnostic output.

For example, by default `armclang` checks the format of `printf()` statements to ensure that the number of `%` format specifiers matches the number of data arguments. The following code generates a warning:

```
printf("Result of %d plus %d is %d\n", a, b);
```

```
armclang --target=aarch64-arm-none-eabi -c hello.c
hello.c:25:36: warning: more '%' conversions than data arguments [-Wformat]
  printf("Result of %d plus %d is %d\n", a, b);
```

To suppress this warning, use `-Wno-format`:

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

Related references

Chapter 7 Coding Considerations on page 7-51.

Related information

The LLVM Compiler Infrastructure Project.
Clang Compiler User's Manual.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

2.3 Options for controlling diagnostics with the other tools

A number of different options control diagnostics with the `armasm`, `armlink`, `armar`, and `fromelf` tools.

The following options control diagnostics:

`--brief_diagnostics`
: `armasm` only. Uses a shorter form of the diagnostic output. In this form, 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.

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
```


Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Chapter 3
Compiling C and C++ Code

Describes how to compile C and C++ code with `armclang`.

It contains the following sections:

- *3.1 Specifying a target architecture, processor, and instruction set* on page 3-29.
- *3.2 Using C++ libraries* on page 3-32.
- *3.3 Using inline assembly code* on page 3-33.
- *3.4 Using intrinsics* on page 3-34.
- *3.5 Preventing the use of floating-point instructions and registers* on page 3-35.
- *3.6 Bare-metal Position Independent Executables* on page 3-36.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

3.1 Specifying a target architecture, processor, and instruction set

When compiling code, the compiler must know which architecture or processor to target, which optional architectural features are available, and which instruction set to use.

Overview

If you only want to run code on one particular processor, you can target that specific processor. Performance is optimized, but code is only guaranteed to run on that processor.

If you want your code to run on a wide range of processors, you can target an architecture. The code runs on any processor implementation of the target architecture, but performance might be impacted.

The options for specifying a target are as follows:

1. Target the execution state (AArch64 or AArch32) using the `--target` option.
2. Target one of the following:
   - an architecture using the `-march` option.
   - a specific processor using the `-mcpu` option.
3. (AArch32 targets only) Specify the floating-point hardware available using the `-mfpu` option, or omit to use the default for the target.
4. (AArch32 targets only) Specify the instruction set using `-marm` or `-mthumb`, or omit to default to `-marm`.

Specifying the target execution state

To specify a target execution state with `armclang`, use the `--target` command-line option:

`--target=`*`arch-vendor-os-abi`*

Supported targets are as follows:

`aarch64-arm-none-eabi`
: Generates A64 instructions for AArch64 state. Implies `-march=armv8-a` unless `-mcpu` is specified.

`arm-arm-none-eabi`
: Generates A32/T32 instructions for AArch32 state. Must be used in conjunction with `-march` (to target an architecture) or `-mcpu` (to target a processor).

Note

The `--target` option is an `armclang` option. For all of the other tools, such as `armasm` and `armlink`, use the `--cpu` and `--fpu` options to specify target processors and architectures.

Note

The `--target` option is mandatory. You must always specify a target execution state.

Specifying the target architecture

Targeting an architecture with `--target` and `-march` generates generic code that runs on any processor with that architecture.

Use the `-march=list` option to see all supported architectures.

Note

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

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Specifying a particular processor

Targeting a processor with `--target` and `-mcpu` optimizes code for the specified processor.

Use the `-mcpu=list` option to see all supported processors.

For ARMv8-A AArch64 targets, you can specify feature modifiers with `-mcpu`, for example `-mcpu=cortex-a57+nocrypto`.

Specifying the floating-point hardware available on the target

The `-mfpu` option overrides the default FPU option implied by the target architecture or processor.

Note

The `-mfpu` option is ignored with ARMv8-A AArch64 targets. Use the `-mcpu` option to override the default FPU for AArch64 targets. For example, to prevent the use of floating-point instructions or floating-point registers for AArch64 targets use the `-mcpu=`*`name`*`+nofp+nosimd` option. Subsequent use of floating-point data types in this mode is unsupported.

Specifying the instruction set

Different architectures support different instruction sets:

- ARMv8-A processors in AArch64 state execute A64 instructions.
- ARMv8-A processors in AArch32 state, as well as ARMv7 and earlier A- and R- profile processors execute A32 (formerly ARM) and T32 (formerly Thumb) instructions.
- M-profile processors execute T32 (formerly Thumb) instructions.

To specify the target instruction set, use the following command-line options:
- `-marm` targets the A32 (formerly ARM) instruction set. This is the default for all targets that support ARM or A32 instructions.
- `-mthumb` targets the T32 (formerly Thumb) instruction set. This is the default for all targets that only support Thumb or T32 instructions.

Note

The `-marm` and `-mthumb` options are not valid with AArch64 targets. The compiler ignores the `-marm` and `-mthumb` options and generates a warning with AArch64 targets.

Command-line examples

The following examples show how to compile for different combinations of architecture, processor, and instruction set:

Table 3-1 Compiling for different combinations of architecture, processor, and instruction set

Architecture	Processor	Instruction set	armclang command
ARMv8-A AArch64 state	Generic	A64	`armclang --target=aarch64-arm-none-eabi test.c`
ARMv8-A AArch64 state	Cortex®-A57	A64	`armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 test.c`
ARMv8-A AArch32 state	Generic	A32	`armclang --target=arm-arm-none-eabi -march=armv8-a test.c`
ARMv8-A AArch32 state	Cortex-A53	A32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-a53 test.c`
ARMv8-A AArch32 state	Cortex-A57	T32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-a57 -mthumb test.c`

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Table 3-1 Compiling for different combinations of architecture, processor, and instruction set (continued)

Architecture	Processor	Instruction set	armclang command
ARMv7-A	Generic	A32	`armclang --target=arm-arm-none-eabi -march=armv7-a test.c`
ARMv7-A	Cortex-A9	A32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-a9 test.c`
ARMv7-A	Cortex-A15	T32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-a15 -mthumb test.c`

Related information

-mcpu.

--target.

-marm.

-mthumb.

-mfpu.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

3.2 Using C++ libraries

There are two C++ libraries available, the libc++ library, and the Rogue Wave Standard C++ library.

Compiling with armclang

When using `armclang` to compile, use the `-stdlib` option to specify the C++ library to add to the system include path.

The options are:

- `libc++` to use the libc++ library. This is the default.
- `legacy_cpplib` to use the Rogue Wave library.

———— **Note** ————

- `armclang` automatically passes the corresponding `--stdlib` option to `armlink`.
- The Rogue Wave C++ libraries are deprecated.

————————————

Linking with armlink

When using `armlink` to link files, use the `--stdlib` option to specify the C++ library to add to the system include path.

The options are:

- `infer`. This is the default.
- `libc++`.
- `legacy_cpplib`.
- `none`.

`armlink` looks at the object files, and if they contain `Lib$Request$$cpplib` it searches `cpplib`, otherwise it searches `libcxx`.

Examples

```
armclang -stdlib=libc++ ...
```

```
armlink --stdlib=infer ...
```

Related information

armclang -stdlib.

armlink --stdlib.

 Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

3.3 Using inline assembly code

The compiler provides an inline assembler that enables you to write optimized assembly language routines, and to access features of the target processor not available from C or C++.

The `__asm` keyword can incorporate inline GCC syntax assembly code into a function. 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 ARM assembler syntax. See the *Migration and Compatibility Guide* for more information about migrating ARM syntax assembly code to GCC syntax.

The general form of an `__asm` inline assembly statement is:

`__asm(`*`code [: output_operand_list [: input_operand_list [: clobbered_register_list]]]`*`);`

`code` is the assembly code. In this example, this is `"ADD %[result], %[input_i], %[input_j]"`.

`output_operand_list` is an optional 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)`.

`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)`.

`clobbered_register_list` is an optional list of clobbered registers. In this example, this is omitted.

Related information

Migrating ARM syntax assembly code to GNU syntax.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

3.4 Using intrinsics

Compiler intrinsics are functions provided by 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 in-built 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, often these operations cannot be efficiently implemented 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 */
}
```

The use of compiler intrinsics offers a number of performance benefits:

- The low-level instructions substituted for an intrinsic might be more efficient than corresponding implementations in C or C++, resulting 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 could not otherwise have performed.

These performance benefits can be significant for real-time processing applications. However, care is required because the use of intrinsics can decrease code portability.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

3.5 Preventing the use of floating-point instructions and registers

You can instruct the compiler to prevent the use of floating-point instructions and floating-point registers.

Floating-point computations and linkage

Floating-point computations can be performed by:

- Floating-point instructions, executed by a hardware coprocessor. The resulting code can only be run on processors with Vector Floating Point (VFP) coprocessor hardware.
- Software library functions, through the floating-point library `fplib`. This library provides functions that can be called to implement floating-point operations using no additional hardware.

Code that uses hardware floating-point instructions is more compact and offers better performance than code that performs floating-point arithmetic in software. However, hardware floating-point instructions require a VFP coprocessor.

Floating-point linkage controls which registers are used to pass floating-point parameters and return values:

- Software floating-point linkage means that the parameters and return values for functions are passed using the ARM integer registers r0 to r3 and the stack. The benefits of using software floating-point linkage include:
  — Code can run on a processor with or without a VFP coprocessor.
  — Code can link against libraries compiled for software floating-point linkage.
- Hardware floating-point linkage uses the VFP coprocessor registers to pass the arguments and return value. The benefit of using hardware floating-point linkage is that it is more efficient than software floating-point linkage, but you must have a VFP coprocessor

Configuring the use of floating-point instructions and registers

When compiling for AArch64 state:

- By default, the compiler uses hardware floating-point instructions and hardware floating-point linkage.
- Use the `-mcpu=`*`name`*`+nofp+nosimd` option to prevent the use of both floating-point instructions and floating-point registers:

  ```
  armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53+nofp+nosimd test.c
  ```

  Subsequent use of floating-point data types in this mode is unsupported.

When compiling for AArch32 state:

- When using `--target=arm-arm-none-eabi`, the compiler uses hardware floating-point instructions and software floating-point linkage. This corresponds to the option `-mfloat-abi=softfp`.
- Use the `-mfloat-abi=soft` option to use software library functions for floating-point operations and software floating-point linkage:

  ```
  armclang --target=arm-arm-none-eabi -march=armv8-a -mfloat-abi=soft test.c
  ```

- Use the `-mfloat-abi=hard` option to use hardware floating-point instructions and hardware floating-point linkage:

  ```
  armclang --target=arm-arm-none-eabi -march=armv8-a -mfloat-abi=hard test.c
  ```

Related information

-mcpu.

-mfloat-abi.

-mfpu.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

3.6 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 but can be executed at any suitably aligned address.

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 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\n");
  return 0;
}
```

To compile and automatically link this code for bare-metal PIE, use the `-fbare-metal-pie` option with `armclang`:

```
armclang -fbare-metal-pie --target=arm-arm-none-eabi -march=armv8-a hello.c -o hello
```

Alternatively, you can compile with `armclang -fbare-metal-pie` and link with `armlink --bare_metal_pie` as separate steps:

```
armclang -fbare-metal-pie --target=arm-arm-none-eabi -march=armv8-a -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 it may be compiled with the option `--apcs=/ropi`.

If you are using link time optimization, use the `armlink --lto-set-relocation-model=pic` option to tell the link time optimizer to produce position independent code:

```
armclang -flto -fbare-metal-pie --target=arm-arm-none-eabi -march=armv8-a -c hello.c -o
hello.bc
armlink --lto --lto_set_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.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

- 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, be aware that 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. 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 supplied by the C library the linker requires that the user provides their own routine for initializing the stack and heap. This user supplied stack and heap routine is run prior to the routine `__arm_relocate_pie_` so it is necessary to ensure that this routine only uses PC relative addressing.

Related information

--fpic.

--pie.

--bare_metal_pie.

--ref_pre_init.

-fbare-metal-pie.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Chapter 4
Assembling Assembly Code

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

It contains the following sections:

- *4.1 Assembling ARM and GNU syntax assembly code* on page 4-39.
- *4.2 Preprocessing assembly code* on page 4-41.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

4.1 Assembling ARM and GNU syntax assembly code

The ARM Compiler 6 toolchain can assemble both ARM and GNU syntax assembly language source code.

ARM and GNU are two different syntaxes for assembly language source code. They are similar, but have a number of differences. For example, ARM 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 ARM and GNU syntax assembly to help you migrate legacy assembly code.

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

ARM syntax assembly:

```
; Simple ARM 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 ARM syntax. Use `armasm` to assemble legacy ARM syntax assembly code. Typically, you invoke the `armasm` assembler as follows:

```
armasm --cpu=8-A.64 -o file.o file.s
```

GNU syntax assembly:

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

        .section .text,"x"
        .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` 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
```


Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Related information

GNU Binutils - Using as.

Migrating ARM syntax assembly code to GNU syntax.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

4.2 Preprocessing assembly code

Assembly code that contains C directives, for example `#include` or `#define`, must be resolved by the C preprocessor prior to assembling.

By default, `armclang` uses the assembly code source file suffix to determine whether or not to run the C preprocessor:

- The `.s` (lower-case) suffix indicates assembly code that does not require preprocessing.
- The `.S` (upper-case) 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 directives and `armclang` must run the C preprocessor. The `-x` option only applies to input files that follow it on the command line.

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 -E test.S
  ```
- Use the `-x assembler-with-cpp` option to tell `armclang` that the assembly source file requires preprocessing.

  For example:

  ```
  armclang --target=arm-arm-none-eabi -march=armv8-a -E -x assembler-with-cpp test.s
  ```

Note

The `-E` option specifies that `armclang` only executes the preprocessor step.

Related information

-x.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Chapter 5
Linking Object Files to Produce an Executable

Describes how to link object files to produce an executable image with `armlink`.

It contains the following sections:

- *5.1 Linking object files to produce an executable* on page 5-43.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

5.1 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
```


Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Chapter 6
Optimization

Describes how to use `armclang` to optimize for either code size or performance, and the impact of the optimization level on the debug experience.

It contains the following sections:

- *6.1 Optimizing for code size or performance* on page 6-45.
- *6.2 Optimizing across modules with link time optimization* on page 6-46.
- *6.3 How optimization affects the debug experience* on page 6-50.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

6.1 Optimizing for code size or performance

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

These 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.

By default, `armclang` does not perform optimization. That is, the default optimization level is `-O0`.

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

`-O0 | -O1 | -O2 | -O3`
: Specify the level of optimization to be used when compiling source files, where `-O0` is the minimum and `-O3` is the maximum.

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

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

`-Os`
: Performs optimizations to reduce the image size at the expense of a possible increase in execution time, balancing code size against code speed.

`-Oz`
: Optimizes for code size.

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

`-flto`
: Enables link time optimization, which lets the linker make additional optimizations across multiple source files.

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 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.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

6.2 Optimizing across modules with link time optimization

Additional optimization opportunities are available at link time, 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 the compiler might miss some optimization opportunities, such as cross-module inlining.

When link time optimization is enabled, the compiler translates source code into an intermediate form called bitcode. At link time, the linker collects all bitcode files together and sends them to the link time optimizer (llvm-lto) as one or more larger units. Collecting modules together means 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.

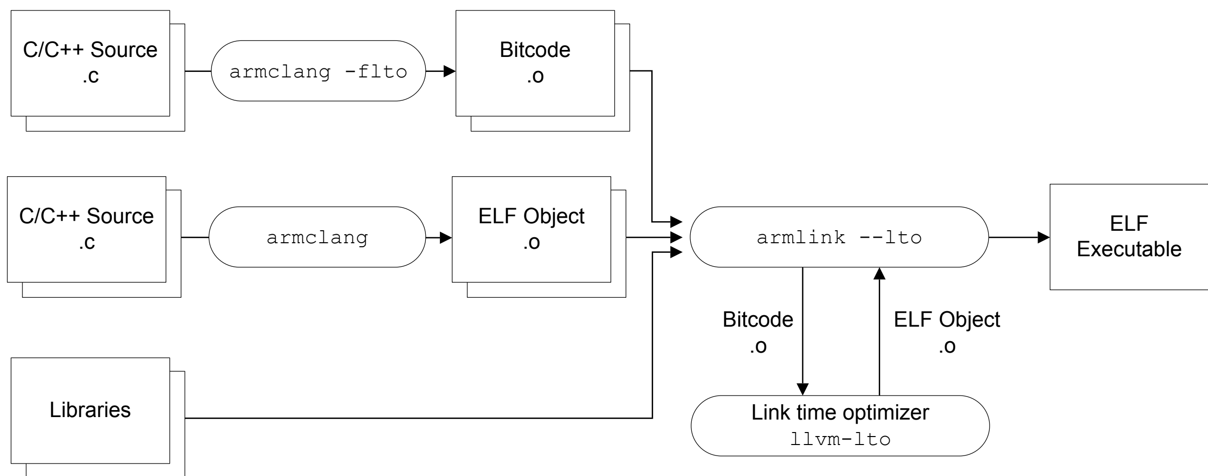


Figure 6-1 Link time optimization

Note

Bitcode files and ELF object files both have the default extension `.o`. You can use `armclang -c -o` *`filename`* to specify a different filename and extension for bitcode files if desired, for example `.bc`.

Note

ARM Compiler does not support link time optimization on 32-bit Red Hat Enterprise Linux platforms.

This section contains the following subsections:
- *6.2.1 Enabling link time optimization* on page 6-46.
- *6.2.2 Restrictions with link time optimization* on page 6-47.

6.2.1 Enabling link time optimization

To enable link time optimization:

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

1. At compilation time, use the `armclang` option `-flto` to produce bitcode files suitable for link time optimization.
2. At link time, use the `armlink` option `--lto` to enable link time optimization for the specified bitcode files.

   Note

   `armclang` automatically passes the `--lto` option to `armlink` if the `-flto` option is used without the `-c` option.

Example 1: Optimizing all source files

The following example performs link time optimization 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 bitcode files `src1.o`, `src2.o`, and `src3.o`.
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.

Example 2: Optimizing a subset of source files

The following example performs link time optimization 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.bc
armclang --target=arm-arm-none-eabi -march=armv8-a -c -flto src3.c -o src3.bc
armlink --lto src1.o src2.bc src3.bc -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 bitcode files `src2.bc` and `src3.bc`.
3. `armlink` passes the bitcode files `src2.bc` and `src3.bc` 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 produced by the link time optimizer to create the executable `output.axf`.

Related references

6.2.2 Restrictions with link time optimization on page 6-47.

Related information

-flto.

--lto armlink option.

6.2.2 Restrictions with link time optimization

The support for link time optimization in ARM Compiler 6 is alpha quality. Future releases may have fewer restrictions and more features. The user interface to link time optimization may be subject to change in future releases.

Red Hat Enterprise Linux platforms

ARM Compiler does not support link time optimization on 32-bit Red Hat Enterprise Linux platforms.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Libraries

`armlink` resolves all link time code generation before libraries are loaded. This allows bitcode files to reference code and data that resides in libraries, however this means that the link time optimizer cannot use any bitcode files in libraries or be aware of references from library objects to bitcode files as the objects from the libraries have not yet been loaded. See *References to symbols in bitcode files from libraries* for more information.

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 bitcode file while loading from a library.

References to symbols in bitcode files from libraries

The link time optimizer may remove global symbols that have not been referenced from other bitcode or other ELF files. If there are symbols that are referenced only from objects in libraries then the link time optimizer might remove them, leading to an undefined symbol error at link time.

For example, consider the following source code:

```
file_in_library.c
-----------------
extern void function(void);

void library_function(void)
{
    function();
}

file.c
------
void function(void)
{
    ...
}

extern void library_function(void);

int main(void)
{
    library_function();
    function();
}
```

The following commands compile the library source code, create a library file, compile the program source code, then link the resulting object files using link time optimization:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -c file_in_library.c
armar --create library.a file_in_library.o
armclang --target=arm-arm-none-eabi -march=armv8-a -c -flto file.c
armlink --lto file.o library.a
```

The link time optimizer might remove `function()` as unused because it is only referenced by a library function. This results in the following error:

```
Error: L6218E: Undefined symbol function (referred from file_in_library.o).
Finished: 0 information, 0 warning and 1 error messages.
```

To prevent the link time optimizer from removing the symbol you must use one or more of the following `armlink` command line options:

- `--keep` *`symbol`* for each symbol in bitcode objects that is referenced only from objects from libraries.
- `--lto_keep_all_symbols` to prevent the optimizer from removing symbols.

Partial linking

The `armlink` options `--partial` and `--ldpartial` only work with ELF files. The linker will give an error message if it detects a bitcode file.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

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 these will be matched by scatter loading selectors as normal.

When all the bitcode files are optimized at once there is just one ELF object file output and this can be named by the `armlink` option `--lto_set_intermediate_filename`. This ELF file name can be referenced in the scatter loading file.

We recommend that link time optimization is only performed on code and data that does not require precise placement in the scatter file, with general scatter loading selectors such as `*(+RO)` and `.ANY(+RO)` used to select sections generated by link time optimization.

It is not possible to match bitcode files by name in scatter loading.

Note

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

Limited compatibility checking

The linker does not perform extensive checks on the bitcode files before passing them to the optimizer. Extra care must be taken to present compatible bitcode files to `armlink`.

Executable and library compatibility

The `armclang` and `llvm-lto` executables and the `libLTO` library must come from the same ARM Compiler 6 installation. Any use of `llvm-lto` or `libLTO` other than those supplied with ARM Compiler 6 is unsupported.

Related references

6.2.1 Enabling link time optimization on page 6-46.

Related information

-flto.

--lto armlink option.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

6.3 How optimization affects the debug experience

There is a trade-off between optimizing code and the debug experience.

The precise optimizations performed by the compiler depend both on the level of optimization chosen, and whether you are optimizing for performance or code size.

The lowest optimization level, `-O0`, provides the best debug experience because the structure of the generated code directly corresponds to the source code.

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 cannot be described by debug information.

Related information

-O.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Chapter 7
Coding Considerations

Describes how you can use programming practices and techniques to increase the portability, efficiency and robustness of your C and C++ source code.

It contains the following sections:

- *7.1 Optimization of loop termination in C code* on page 7-52.
- *7.2 Loop unrolling in C code* on page 7-54.
- *7.3 Compiler optimization and the volatile keyword* on page 7-56.
- *7.4 Stack use in C and C++* on page 7-58.
- *7.5 Methods of minimizing function parameter passing overhead* on page 7-60.
- *7.6 Inline functions* on page 7-61.
- *7.7 Integer division-by-zero errors in C code* on page 7-62.
- *7.8 Infinite Loops* on page 7-64.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

7.1 Optimization of loop termination in C code

Loops are a common construct in most programs. Because a significant amount of execution time is often spent in loops, it is worthwhile paying attention to time-critical loops.

The loop termination condition can cause significant overhead if written without caution. Where possible:

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

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 7-1 C code for incrementing and decrementing loops

Incrementing loop

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

Decrementing loop

```
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 of the machine code produced by `armclang -Os -S --target=arm-arm-none-eabi -march=armv8-a` for each of the sample implementations above.

Table 7-2 C disassembly for incrementing and decrementing loops

Incrementing 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
```

Decrementing loop

```
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.

In addition to saving an instruction in the loop, 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. This eases register allocation. It is even more important if the original termination condition involves a function call. For example:

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


Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

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

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

7.2 Loop unrolling in C code

Loops are a common construct in most programs. Because a significant amount of execution time is often spent in loops, it is worthwhile paying attention to time-critical loops.

Small loops can be unrolled for higher performance, with the disadvantage of increased code size. When a loop is unrolled, the loop counter requires updating less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled so that the loop overhead completely disappears. The compiler unrolls loops automatically at `-O3 -Otime`. Otherwise, any unrolling must be done in source code.

Note

Manual unrolling of loops might hinder the automatic re-rolling of loops and other loop optimizations by the compiler.

The advantages and disadvantages of loop unrolling can be illustrated using the two sample routines shown in the following table. Both routines efficiently test a single bit by extracting the lowest bit and counting it, after which the bit is shifted out.

The first implementation uses a loop to count bits. The second routine is the first implementation unrolled four times, with an optimization applied by combining the four shifts of `n` into one shift.

Unrolling frequently provides new opportunities for optimization.

Table 7-3 C code for rolled and unrolled bit-counting loops

Bit-counting loop

```
int countbit1(unsigned int n)
{
    int bits = 0;
    while (n != 0)
    {
        if (n & 1) bits++;
        n >>= 1;
    }
    return bits;
}
```

Unrolled bit-counting loop

```
int countbit2(unsigned int n)
{
    int bits = 0;
    while (n != 0)
    {
        if (n & 1) bits++;
        if (n & 2) bits++;
        if (n & 4) bits++;
        if (n & 8) bits++;
        n >>= 4;
    }
    return bits;
}
```

The following table shows the corresponding disassembly of the machine code produced by the compiler for each of the sample implementations above, where the C code for each implementation has been compiled using `armclang -Os -S --target=arm-arm-none-eabi -march=armv8-a`.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Table 7-4 Disassembly for rolled and unrolled bit-counting loops

Bit-counting loop	Unrolled bit-counting loop

<pre>countbit1:
        mov   r1, r0
        mov   r0, #0
        cmp   r1, #0
        bxeq   lr
        mov   r2, #0
.LBB0_1:
        and   r3, r1, #1
        cmp   r2, r1, lsr #1
        add   r0, r0, r3
        lsr   r3, r1, #1
        mov   r1, r3
        bne   .LBB0_1
        bx   lr</pre>
	<pre>countbit2:
	        mov r1, r0
	        mov r0, #0
	        cmp r1, #0
	        bxeq lr
	        mov r2, #0
	.LBB1_1:
	        and r3, r1, #1
	        cmp r2, r1, lsr #4
	        add r0, r0, r3
	        ubfx r3, r1, #1, #1
	        add r0, r0, r3
	        ubfx r3, r1, #2, #1
	        add r0, r0, r3
	        ubfx r3, r1, #3, #1
	        add r0, r0, r3
	        lsr r3, r1, #4
	        mov r1, r3
	        bne .LBB1_1
	        bx lr</pre>

The unrolled version of the bit-counting loop is faster than the original version, but has a larger code size.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

7.3 Compiler optimization and the volatile keyword

Higher optimization levels can reveal problems in some programs that are not apparent at lower optimization levels, for example, missing **volatile** qualifiers.

This can manifest itself in a number of ways. Code might become stuck in a loop while polling hardware, multi-threaded code might exhibit strange behavior, or optimization might result in the removal of code that implements deliberate timing delays. In such cases, it is possible that some variables are required to be declared as **volatile**.

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, or by hardware. Because the value of a **volatile**-qualified variable can change at any time, the actual variable in memory must always be accessed whenever the variable is referenced in code. This means the compiler cannot perform optimizations on the variable, for example, caching its value in a register to avoid memory accesses. Similarly, when used in the context of implementing a sleep or timer delay, declaring a variable as **volatile** tells the compiler that a specific type of behavior is intended, and that such code must not be optimized in such a way that it removes the intended functionality.

In contrast, when a variable is not declared as **volatile**, the compiler can assume its value cannot be modified in unexpected ways. Therefore, the compiler can perform optimizations on the variable.

The use of the **volatile** keyword is illustrated in the two sample routines in the following table. Both of these routines read a buffer 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 two versions of the routine differ only in the way that `buffer_full` is declared. The first routine version is incorrect. Notice that the variable `buffer_full` is not qualified as **volatile** in this version. In contrast, the second version of the routine shows the same loop where `buffer_full` is correctly qualified as **volatile**.

Table 7-5 C code for nonvolatile and volatile buffer loops

Nonvolatile version of buffer loop

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

Volatile version of buffer loop

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

The following table shows the corresponding disassembly of the machine code produced by the compiler for each of the examples above, where the C code for each implementation has been compiled using `armclang -Os -S --target=arm-arm-none-eabi -march=armv8-a`.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Table 7-6 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 version of the buffer loop in the above table, 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 contrast, in the disassembly of the volatile version of the buffer loop, the compiler assumes the value of `buffer_full` can change outside the program and performs no optimizations. Consequently, the value of `buffer_full` is loaded into register `r2` inside the loop labeled `.LBB1_1`. As a result, the loop `.LBB1_1` is implemented correctly in assembly code.

To avoid optimization problems caused by changes to program state external to the implementation, you must declare variables as **`volatile`** whenever their values can change unexpectedly in ways unknown to the implementation.

In practice, you must declare a variable as **`volatile`** whenever you are:

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

The compiler does not optimize the variables you have declared as volatile.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

7.4 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 for the ARM 64-bit Architecture* (AAPCS64), for instance, when register contents are saved on entry into subroutines.
- Local variables, including local arrays, structures, unions, and in C++, classes.

Some stack usage is not obvious, such as:

- Local integer or floating point variables are allocated stack memory if they are spilled (that is, not allocated to a register).
- Structures are normally allocated to the stack. A space equivalent to `sizeof(struct)` padded to a multiple of 16 bytes is reserved on the stack. The compiler tries 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 16 bytes is reserved on the stack.

  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.
- Generally, code compiled for processors that support only 16-bit encoded Thumb® instructions makes more use of the stack than A64 code, ARM code and code compiled for processors that support 32-bit encoded Thumb instructions. This is because 16-bit encoded Thumb instructions have only eight registers available for allocation, compared to fourteen for ARM code and 32-bit encoded Thumb instructions.
- The AAPCS64 requires that some function arguments are passed through the stack instead of the registers, depending on their type, size, and order.

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:

- Link with `--callgraph` to produce a static callgraph. This shows information on all functions, including stack use.

  This uses DWARF frame information from the `.debug_frame` section. Compile with the `-g` option to generate the necessary DWARF information.
- Link with `--info=stack` or `--info=summarystack` to list the stack usage of all global symbols.
- Use the debugger to set a watchpoint on the last available location in the stack and see if the watchpoint is ever hit.
- Use the 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 found by the static analysis. Try to generate interrupts where appropriate, so that they are included in the stack trace.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

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 Fixed Virtual Platforms (FVP), and define a region of memory where access is not allowed directly below your stack in memory, with a map file. If the stack overflows into the forbidden region, a data abort occurs, which can be trapped by the debugger.

Methods of reducing stack usage

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

- Writing small functions that only require a small number of variables.
- Avoiding the use of large local structures or arrays.
- Avoiding recursion, for example, by using an alternative algorithm.
- Minimizing the number of variables that are in use at any given time at each point in a function.
- Using C block scope and declaring variables only where they are required, so overlapping the memory used by distinct scopes.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

7.5 Methods of minimizing function parameter passing overhead

There are a number of 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 no more than one fewer argument than the efficient limit, because of the implicit `this` pointer argument that 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, so that the cost of passing the stacked arguments is outweighed.
- Put related arguments in a structure, and pass a pointer to the structure in any function call. This reduces the number of parameters and increases readability.
- For 32-bit architectures, minimize the number of `long long` parameters, because these take two argument words that have to be aligned on an even register index.
- For 32-bit architectures, minimize the number of `double` parameters when using software floating-point.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

7.6 Inline functions

Inline functions offer a trade-off between code size and performance. By default, the compiler decides for itself whether to inline code or not.

See the Clang documentation for more information about inline functions.

Related information

Language Compatibility.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

7.7 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);
```


Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

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.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

7.8 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.

How to write an infinite loop in `armclang`

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.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Chapter 8
Using the ARMv8-M Security Extensions [BETA]

Describes how to use the ARMv8-M Security Extensions in your applications.

Note

This chapter describes *[BETA]* on page 1-13 features.

It contains the following sections:

- *8.1 Overview of ARMv8-M Security Extensions [BETA]* on page 8-66.
- *8.2 Creating a custom import library [BETA]* on page 8-74.
- *8.3 Generating a secure gateway veneer for a secure image [BETA]* on page 8-75.
- *8.4 Using an import library when building a non-secure image [BETA]* on page 8-78.

 Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

8.1 Overview of ARMv8-M Security Extensions [BETA]

ARMv8-M Security Extensions defines a system-wide division of physical memory into secure regions and non-secure regions and two system-wide security states that are enforced by hardware.

Note

This topic describes a *[BETA]* on page 1-13 feature.

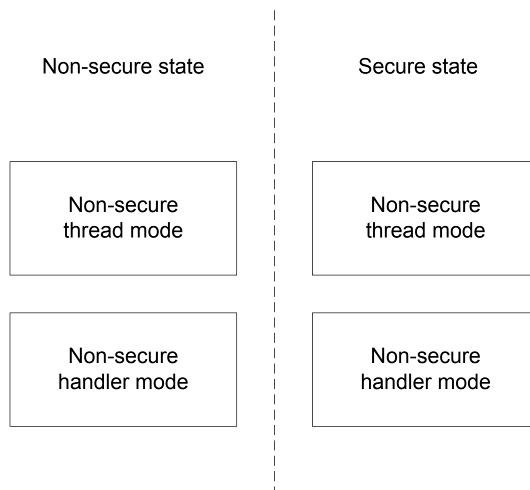
There is a direct relationship between the memory regions and the security states:

- Code executed from a non-secure region is executed in Non-secure state and can only access memory in non-secure regions.
- Code executed from a secure region is executed in Secure state and can access memory in both secure and non-secure regions.

Note

Floating-point code is not supported in secure code.

The security states are orthogonal to the exception level, as shown in the following figure:



Memory regions can be defined by the system through the *Implementation Defined Attribute Unit* (IDAU) or can be controlled in software through the memory mapped *Security Attribute Unit* (SAU) registers.

Parts of the system are banked between the security states. The stack pointer is banked, resulting in a stack pointer for each combination of security state and exception level. All parts of the system accessible in Non-secure state can be accessed in Secure state, including the banked parts.

This section contains the following subsections:

- *8.1.1 Security state changes [BETA]* on page 8-66.
- *8.1.2 The ARMv8-M TT instruction [BETA]* on page 8-67.
- *8.1.3 Requirements for creating secure code [BETA]* on page 8-68.
- *8.1.4 Executable files in Secure and Non-secure states [BETA]* on page 8-69.
- *8.1.5 Secure gateway [BETA]* on page 8-70.
- *8.1.6 Entry functions [BETA]* on page 8-72.
- *8.1.7 Non-secure function call [BETA]* on page 8-73.
- *8.1.8 Non-secure function pointer [BETA]* on page 8-73.

8.1.1 Security state changes [BETA]

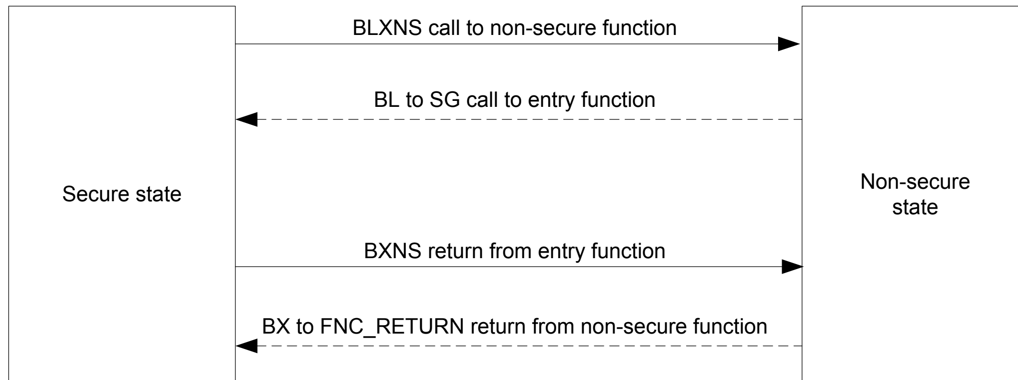
The system boots in Secure state and can change security states using branches.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Note

This topic describes a *[BETA]* on page 1-13 feature.

The security state changes are shown in the following figure:



Secure to Non-secure state

Transitions from Secure to Non-secure state can be initiated by software through the use of the `BXNS` and `BLXNS` instructions that have the *Least Significant Bit* (LSB) of the target address unset. The reserved value `FNC_RETURN` is written to `lr` when executing the `BLXNS` instruction.

Non-secure to Secure state

Transitions from Non-secure to Secure state can be initiated by software in the following ways:

- A `BL` branch to a secure gateway.
- A `BX` branch to the reserved value `FNC_RETURN`.

Security state transitions caused by hardware through the handling of interrupts are transparent to software.

Related information

BLX, BLXNS instruction.

BX, BXNS instruction.

8.1.2 The ARMv8-M TT instruction [BETA]

The ARMv8-M architecture includes the Test Target instruction `TT`. It takes a memory address and returns the configuration of the *Memory Protection Unit* (MPU) at that address. ARM Compiler 6 provides intrinsics to generate the `TT` instruction.

When executed in the Secure state the result of this instruction is extended to return the SAU and IDAU configurations at the specific address.

The MPU is banked between the two security states. The optional `A` flag makes the `TT` instruction read the MPU of the Non-secure state when the `TT` instruction is executed from the Secure state.

Use the `TT` instruction to check the access permissions that different security states and privilege levels have on memory at a specified address.

Using the TT instruction in C code [BETA]

To make use of the ARMv8-M `TT` instruction in C code, you must include the `<arm_cmse.h>` header. This header defines the intrinsics and the return types you can use in your code.

Note

This topic describes a *[BETA]* on page 1-13 feature.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

The value of the `__ARM_FEATURE_CMSE` predefined macro indicates the availability of the `TT` instruction.

Related information

TT instruction intrinsics.
Predefined macros.
TT, TTT, TTA, TTAT instruction.

8.1.3 Requirements for creating secure code [BETA]

To prevent secure code and data from being accessed from Non-secure state, secure code must meet certain minimum requirements.

Note

Floating-point code is not supported in secure code.

Information leakage

Information leakage from the Secure state to the Non-secure state might occur through parts of the system that are not banked between the security states. The unbanked registers that are accessible by software are:

- General purpose registers except for the stack pointer (R0-R12, R14-R15).
- Floating point registers (S0-S31, D0-D15).
- The N, Z, C, V, Q, and GE bits of the APSR register.
- The FPSCR register.

Secure code must clear secret information from unbanked registers before initiating a transition from Secure to Non-secure state.

Non-secure memory access

When secure code needs to access Non-secure memory using an address calculated by the Non-secure state, it cannot trust that the address lies in a Non-secure memory region. Also, the MPU is banked between the Security states. Therefore, Secure and Non-secure code might have different access rights to Non-secure memory.

Secure code that accesses Non-secure memory on behalf of the Non-secure state must only do so if the Non-secure state has permission to perform the same access itself.

The secure code can use the `TT` instruction to check Non-secure memory permissions.

Secure code must not access Non-secure memory unless it does so on behalf of the Non-secure state.

Data belonging to Secure code must reside in Secure memory.

Volatility of non-secure memory

Non-secure memory can change asynchronously to the execution of Secure code. There are two causes:

- Interrupts handled in Non-secure state can change Non-secure memory.
- The debug interface can be used to change Non-secure memory.

There can be unexpected consequences when Secure code accesses Non-secure memory:

```
int array[N]
void foo(int *p) {
    if (*p >= 0 && *p < N) {
        // non-secure memory (*p) is changed at this point
        array[*p] = 0;
    }
}
```

When the pointer `p` points to Non-secure memory, it is possible for its value to change after the memory accesses used to perform the array bounds check, but before the memory access used to index the array. Such an asynchronous change to Non-secure memory renders this array bounds check useless.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Secure code must handle Non-secure memory as volatile.

You can handle the case shown in the previous example as follows:

```
int array[N]
void foo(volatile int *p) {
    int i = *p;
    if (i >= 0 && i < N) {
        array[i] = 0;
    }
}
```

Inadvertent secure gateway

A Secure Gateway `SG` instruction can occur inadvertently. This can happen in the following cases:

- Uninitialized memory.
- General data in executable memory, for example jump tables.
- A 32-bit wide instruction that contains the bit pattern `0b1110100101111111` in its first half-word that follows an `SG` instruction, for example two successive `SG` instructions.
- A 32-bit wide instruction that contains the bit pattern `0b1110100101111111` in its last half-word that is followed by an `SG` instruction, for example an `SG` instruction that follows an `LDR` (immediate) instruction.

If an inadvertent `SG` instruction occurs in an NSC region, the result is an inadvertent secure gateway.

Memory in an NSC region must not contain an inadvertent `SG` instruction.

The secure gateway veneers limit the instructions that need to be placed in NSC regions. If the NSC regions contain only these veneers, an inadvertent secure gateway cannot occur.

Related information

SG instruction.
TT, TTT, TTA, TTAT instruction.

8.1.4 Executable files in Secure and Non-secure states [BETA]

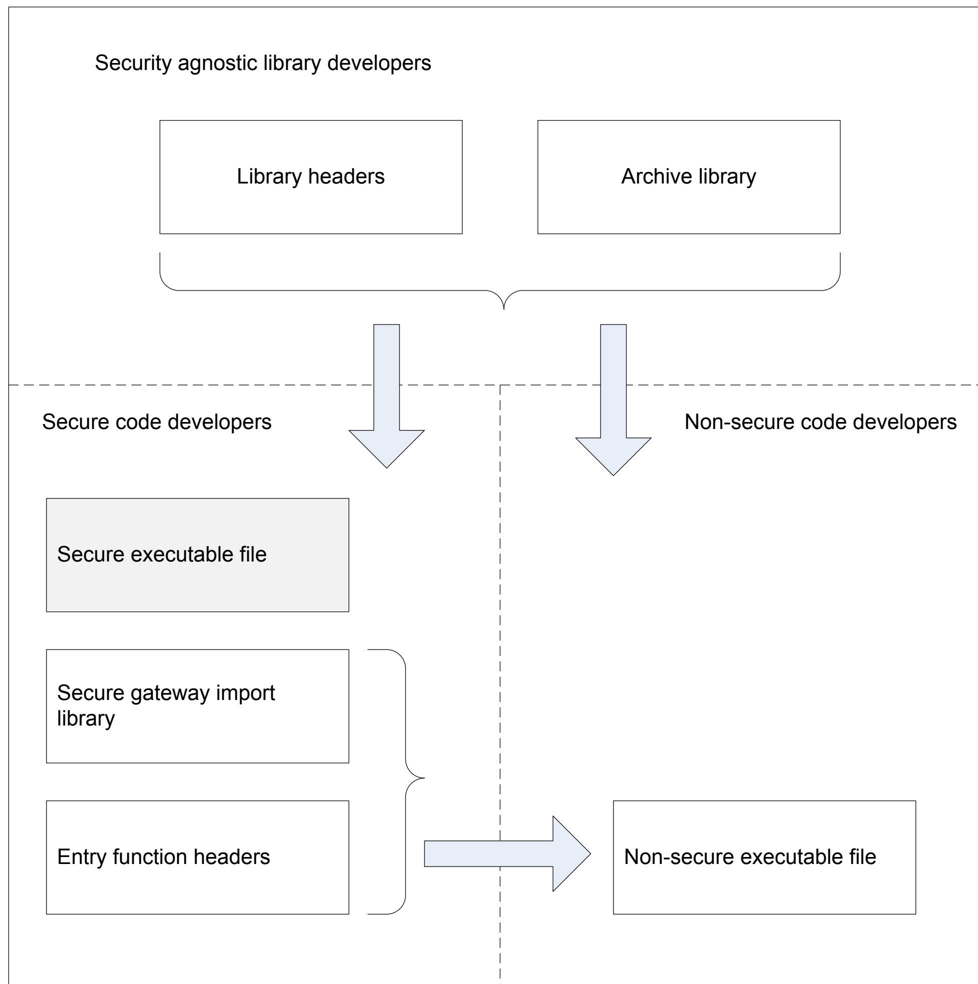
There are two different types of executable files, one for each security state. The Secure state executes secure code from a secure executable file. The Non-secure state executes non-secure code from a non-secure executable file. You can develop the secure and non-secure executable files independently of each other.

A non-secure executable is unaware of security states.

From the point of view of the Non-secure state, a call to a secure gateway is a regular function call, as is the return from a non-secure function call. Therefore, you can develop a non-secure executable file using a toolchain that is not aware of ARMv8-M Security Extensions.

The following figure shows the interaction between developers of secure code, non-secure code, and optionally, security agnostic library code:

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential



8.1.5 Secure gateway [BETA]

A secure gateway is an occurrence of the Secure Gateway instruction (`SG`) in a special type of secure region, named a *Non-Secure Callable* (NSC) region.

When branching to a secure gateway from Non-secure state, the `SG` instruction switches to the Secure state and clears the LSB of the return address in `lr`. In any other situation the `SG` instruction does not change the security state or modify the return address.

The secure gateway import library [BETA]

The secure gateway import library is an object file that contains a symbol table. Each symbol then specifies an absolute address of a secure gateway veneer for an entry function of the same name as the symbol. The non-secure code links against this import library to use the functionality provided by the secure code.

The secure gateway import library is referred to as the *import library* throughout the ARM Compiler documentation.

The ARM Compiler tools provide options to specify the import library:

Input import library

The input import library currently allows you to specify only addresses of entry points that require a veneer to be generated by `armlink`. It does not allow you to specify where manually created secure gateways are to be placed.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

Output import library

The output import library contains all secure gateways. This means the input and output libraries are different if you use manually created secure gateways.

Secure gateway veneers [BETA]

Secure gateway veneers decouple the addresses of secure gateways, in NSC regions, from the rest of the secure code.

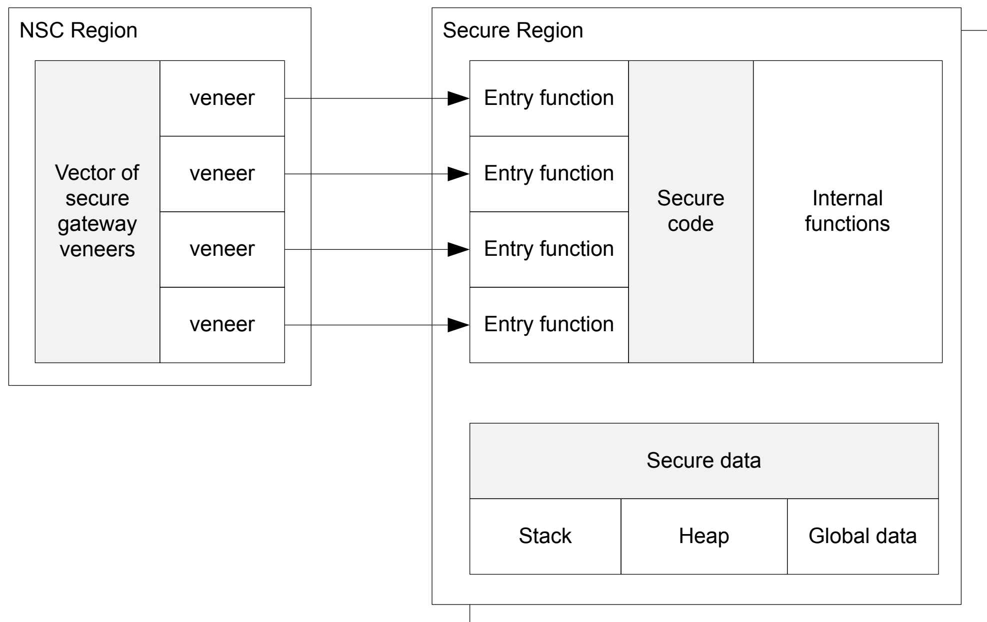
By maintaining a vector of secure gateway veneers at a permanently-fixed address, the rest of the Secure code can be updated independently of Non-secure code. This also limits the amount of code in NSC regions that is called from the Non-secure state.

Vectors of secure gateway veneers are expected to be placed in NSC memory. All other code in the secure executable is expected to be placed in secure memory regions. The ARM Compiler tools allow you to control their placement.

You must take care that the code or data before the vector of secure gateway veneers does not create an inadvertent secure gateway with the first secure gateway veneer in the vector. Although you can control the placement of the secure gateway veneers in a vector, ARM recommends placing the vector of secure gateway veneers at the start of an NSC region.

A vector of secure gateway veneers must be aligned to a 32-byte boundary, and must be zero padded to a 32-byte boundary.

The following figure shows the memory layout of a secure executable:



Related tasks

8.2 Creating a custom import library [BETA] on page 8-74.
8.3 Generating a secure gateway veneer for a secure image [BETA] on page 8-75.
8.4 Using an import library when building a non-secure image [BETA] on page 8-78.

Related information

SG instruction.
Generation of security gateway veneers.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

8.1.6 Entry functions [BETA]

An entry function is a function in secure code that can be called from Secure state, and from Non-secure state through its secure gateway.

You declare an entry function with the attribute `__attribute__((cmse_nonsecure_entry))`.

If an entry function has static linkage, the tools generate a diagnostic message.

An entry function has two ELF function (`STT_FUNC`) symbols to label it:

- A symbol that conforms to the standard naming for C entities defined in *ELF for the ARM® Architecture* is used to label the inline secure gateway of the function if it has one, otherwise the first instruction of the function labels the secure gateway.
- A special symbol that prefixes the standard function name with `__acle_se_` labels the first non-`SG` instruction of the function.

The special symbol acts as an entry function attribute in the relocatable file. Tools that operate on relocatable files can use this symbol to detect the need to generate a secure gateway veneer and a symbol in the import library.

The ARM Compiler toolchain generates a secure gateway veneer for an entry function that has both its symbols labeling the same address. Otherwise a secure gateway is assumed to be present.

To summarize, for a function symbol `foo`:

- A secure gateway veneer for `foo` is only generated if `foo == __acle_se_foo`.
- The symbol foo is copied to the import library if `__acle_se_foo` is present and `foo != __acle_se_foo`.

The address of an entry function must be the address labeled by its standard symbol.

This must be the address of its associated `SG` instruction, usually the first instruction of its secure gateway veneer. This veneer is labeled by the standard symbol name of the function.

You can use the `cmse_nonsecure_caller()` intrinsic to determine if an entry function is called from Secure or Non-secure state.

Arguments and return value

If a compiler is not aware that it is calling a secure entry function it always uses the non-secure stack.

Return from an entry function

Return from an entry function is handled automatically by the compiler if the function is declared an entry function with the attribute `_attribute_((cmse_nonsecure_entry))`.

In assembly, an entry function must use the `BXNS` instruction to return to its non-secure caller.

This instruction switches to Non-secure state if the target address has its LSB unset. The `SG` instruction automatically clears LSB of the return address in LR when it switches the state from Non-secure to Secure.

The code sequence directly preceding the `BXNS` instruction that transitions to non-secure code must:

- Clear all caller-saved registers except:
  - — Registers that hold the result value and the return address of the entry function.
  - — Registers that do not contain secret information.
- Clear all registers and flags that have undefined values at the return of a procedure, according to the *Procedure Call Standard for the ARM® Architecture*.
- Restore all callee-saved registers as mandated by the *Procedure Call Standard for the ARM® Architecture*.

Related information

__attribute__((cmse_nonsecure_entry)) function attribute.

Non-secure function pointer intrinsics.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

8.1.7 Non-secure function call [BETA]

A non-secure function call is a function call from Secure state to the Non-secure state. A non-secure function call can only happen through function pointers. This is a consequence of separating secure and non-secure code into separate executable files.

You must declare a non-secure function type using the function attribute `__attribute__((cmse_nonsecure_call))`.

You must use a non-secure function type only as a base type of a pointer, not as a function definition. This ensures a secure executable file only contains secure function definitions.

Performing a call

A function call through a pointer with a non-secure function type as its base type switches to the Non-secure state. A function call that switches to the Non-secure state, clears LSB of the function address and branches using the `BLXNS` instruction.

All registers that contain secret information are cleared to prevent information leakage when performing a Non-secure function call. Registers that contain values that are used after the non-secure function call are restored after the call returns. Secure code cannot depend on the Non-secure state to restore these registers.

Related information

__attribute__((cmse_nonsecure_call)) function attribute.

8.1.8 Non-secure function pointer [BETA]

A non-secure function pointer is a function pointer that has its LSB unset. It allows you to test the runtime security state that is targeted when performing a call through this pointer. A non-secure function pointer is not a type and must not be confused with the non-secure function type.

Example

The following example shows the use of a non-secure function pointer to share a single variable for secure function pointers and non-secure function pointers:

```
#include <arm_cmse.h>
typedef void __attribute__((cmse_nonsecure_call)) nsfunc(void);
void default_callback(void) { … }

// fp can point to a secure function or a non-secure function
nsfunc *fp = (nsfunc *) default_callback;           // secure function pointer

void __attribute__((cmse_nonsecure_entry)) entry(nsfunc *callback) {
    fp = cmse_nsfptr_create(callback);  // non-secure function pointer
}

void call_callback(void) {
    if (cmse_is_nsfptr(fp)) fp();       // non-secure function call
    else ((void (*)(void)) fp)();       // normal function call
}
```

The global variable `fp` is a non-secure function type but can hold the address of a secure or non-secure function. By using the non-secure function pointer related intrinsics, you can check at runtime the function call to perform:

- `cmse_nsfptr_create(p)`.
- `cmse_is_nsfptr(p)`.

ARM does not recommended sharing such a variable.

Related information

__attribute__((cmse_nonsecure_call)) function attribute.
__attribute__((cmse_nonsecure_entry)) function attribute.
Non-secure function pointer intrinsics.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

8.2 Creating a custom import library [BETA]

You can create your own import library that specifies the entry points that require a veneer generated by `armlink`.

Note

This topic describes a *[BETA]* on page 1-13 feature.

Procedure

1. Create an assembler program, `importlib.s`, that specifies the entry points:

```
        .global entry1
        .type entry1, STT_FUNC
entry1=0x1001

        .global entry2
        .type entry2, STT_FUNC
entry2=0x1009
```

2. Compile the assembler program:

```
$ armclang --target=arm-arm-none-abi -march=armv8-m.main -c importlib.s
```

Related concepts

8.1 Overview of ARMv8-M Security Extensions [BETA] on page 8-66.

Related information

-march armclang option.

--target armclang option.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

8.3 Generating a secure gateway veneer for a secure image [BETA]

`armclang` and `armlink` provide command-line options to generate secure gateway veneers for secure images. This procedure describes how to use the options.

Prerequisites

Note

This topic describes a *[BETA]* on page 1-13 feature.

The following procedure assumes that you have a C program that you can use to create a secure image, `secure.c`.

Procedure

1. Create an interface that is visible to non-secure code in `myinterface.h`, and contains the following:

```
int entry1(int x);
int entry2(int x);
```

2. In the C program for your secure code, `secure.c`, include the following:

```
#include <arm_cmse.h>
#include "myinterface.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 can only be called by secure code.

3. Create an object file using the `armclang -mcmse -mcpu=none` command-line options:

```
$ armclang -target arm-arm-none-eabi -march=armv8-m.main -mcmse -mcpu=none secure.c -o
secure.o
```

The C code in the previous step might produce the following assembly:

```
    .text
    .thumb

func1:
    BX lr
entry1:
__acle_se_entry1:
    PUSH {r11, lr}
    BL func1
    POP {r11, lr}
    BXNS lr
entry2:
__acle_se_entry2:
    PUSH {r11, lr}
    BL entry1
    POP {r11, lr}
    BXNS lr
.weak entry1, entry2
main:
        MOVS r0, #0
        BX lr
```

An entry function does not start with an `SG` instruction but has two symbols labeling its start. This indicates an entry function to the linker.

Note

As an alternative, the compiler can use the `__acle_se_entry1` symbol rather than the `entry1` symbol in function `entry2`. This makes the function call skip the secure gateway veneer.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

When you link the relocatable file corresponding to this assembly code into an executable file, the linker creates the following veneers in a section containing only entry veneers:

```
    .section Veneeer$$CMSE, "ax"
    .thumb
entry1:
    SG
    B.W __acle_se_entry1
entry2:
    SG
    B.W __acle_se_entry2
```

The section with the veneers is aligned on a 32-byte boundary and padded to a 32-byte boundary. You can control the placement of the section with the veneers using a scatter file and place it in your NSC memory region.

4. Create a scatter file containing the `Veneer$$CMSE` selector to place the entry function veneers, for example:

```
LR1
{
   ...
}

...

LOAD_NSCR 0x100 0x1000
{
    EXEC_NSCR 0x100 0x1000
    {
        *(Veneer$$CMSE)
    }
}
...
```

5. Link the object file using the `armlink --fpu=softvfp`, `--import-cmse-lib-out`, and `--import-cmse-lib-in` command-line options and the preprocessed scatter file to create the secure image:

```
$ armlink secure.o -o secure.axf --fpu=softvfp --import-cmse-lib-out importlib.o --
import-cmse-lib-in oldimportlib.o --scatter secure.scf
```

In addition to the final executable, the link in this example also produces the import library, `importlib.o`, for non-secure code. Assuming that the section with veneers is placed at address `0x100`, 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	0x101
STB_GLOBAL, SHN_ABS, STT_FUNC	entry2	0x109

6. Finally, you can:
   a. Pre-load the secure executable file on your device.
   b. Deliver your device with the pre-loaded executable, the import library, and the header file to a party who develops non-secure code for this device.

Related concepts

8.1 Overview of ARMv8-M Security Extensions [BETA] on page 8-66.

Related tasks

8.2 Creating a custom import library [BETA] on page 8-74.

Related information

-march armclang option.
-mcmse armclang option.
-mfpu armclang option.
--target armclang option.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

--fpu linker option.

--import_cmse_lib_in armlink option.

--import_cmse_lib_out armlink option.

--predefine armlink option.

--scatter armlink option.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential

8.4 Using an import library when building a non-secure image [BETA]

An import library contains the addresses of the secure gateway veneers that a non-secure image uses. This is sufficient for the non-secure image to use the secure functionality without having to know anything about the security extensions.

Prerequisites

Note

This topic describes a *[BETA]* on page 1-13 feature.

The following procedure assumes that you have:
- A C program that you can use to create a non-secure image, `nonsecure.c`.
- An input import library, `importlib.o`, that contains the secure gateway veneers.

Procedure

1. Create an object file, `nonsecure.o`:

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

2. Link the object file using the scatter file to create the non-secure image, :

```
$ armlink nonsecure.o importlib.o -o nonsecure.axf --scatter nonsecure.scf
```

Related concepts

8.1 Overview of ARMv8-M Security Extensions [BETA] on page 8-66.

Related tasks

8.2 Creating a custom import library [BETA] on page 8-74.

Related information

-march armclang option.
--target armclang option.
--scatter armlink option.

Copyright © 2014, 2015 ARM. All rights reserved.
Non-Confidential