ARM® Compiler

Version 6.6

Software Development Guide

ARM DUI0773H

ARM® Compiler

Software Development Guide

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
E	24 February 2016	Non-Confidential	ARM Compiler v6.4 Release
F	29 June 2016	Non-Confidential	ARM Compiler v6.5 Release
G	04 November 2016	Non-Confidential	ARM Compiler v6.6 Release
H	08 May 2017	Non-Confidential	ARM Compiler v6.6.1 Release

Non-Confidential Proprietary Notice

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

Contents
ARM® Compiler Software Development Guide

List of Figures
ARM® Compiler Software Development Guide

List of Tables
ARM® Compiler Software Development Guide

Preface

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

It contains the following:

- *About this book* on page 9.

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 Mapping code and data to target memory

Describes how to place your code and data into the correct areas of memory on your target hardware.

Chapter 9 Building Secure and Non-secure Images Using ARMv8-M Security Extensions

Describes how to use the ARMv8-M Security Extensions to build a secure image, and how to allow a non-secure image to call a secure image.

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.

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

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

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

`<and>`
: Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
: Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

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

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

Feedback on content

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

- The title *ARM Compiler Software Development Guide*.
- The number ARM DUI0773H.
- 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*.

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

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.

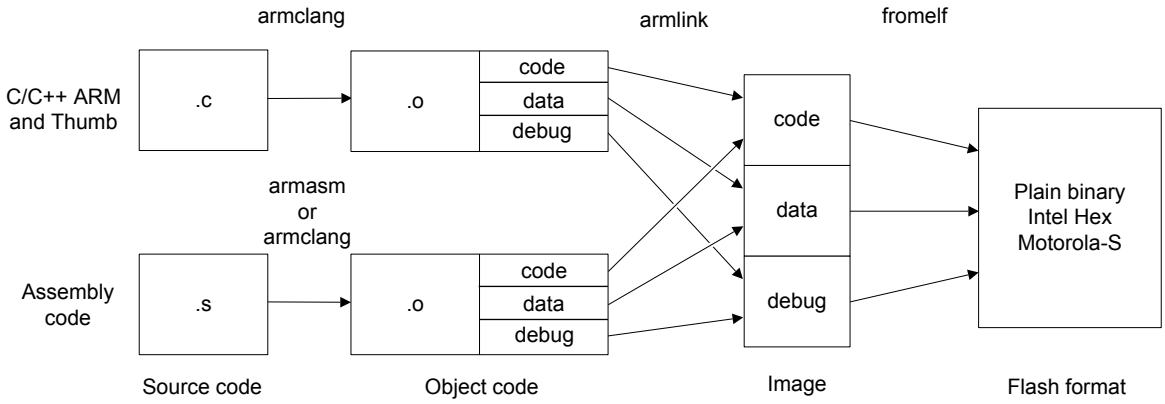


Figure 1-1 Compiler toolchain

The ARM Compiler toolchain comprises the following tools:

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

Note

Disassembly is generated in ARM assembler syntax and not GNU assembler syntax.

Related tasks

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

Related references

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

1.2 Support level definitions

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

ARM Compiler 6 is built on Clang and LLVM technology 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 are 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*.

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:

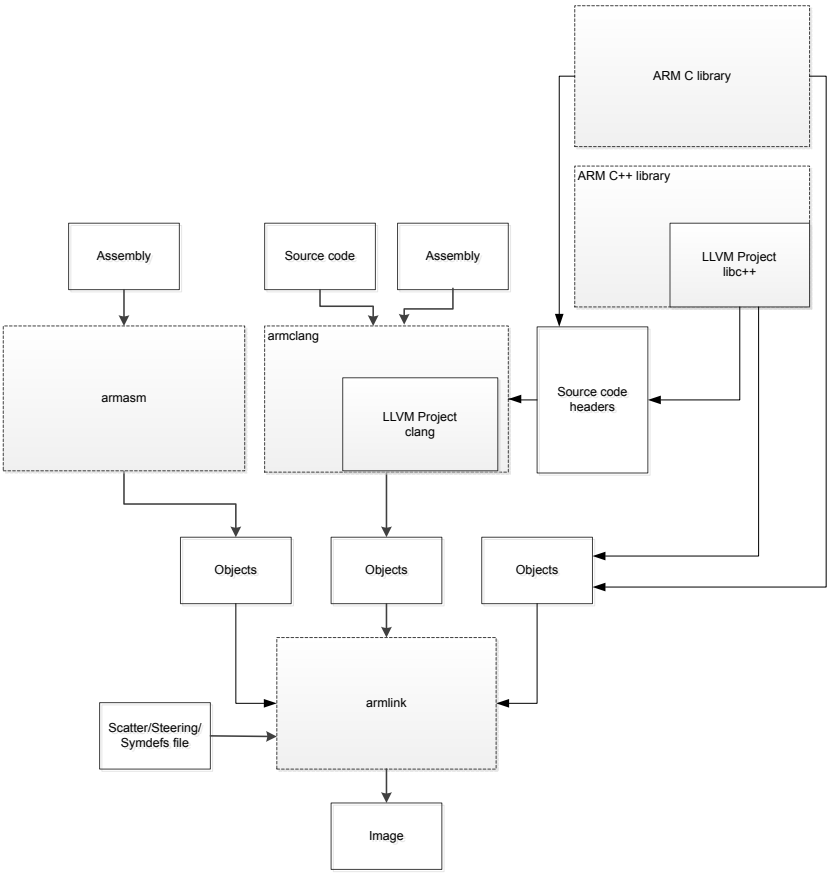


Figure 1-2 Integration boundaries in ARM Compiler 6.

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

Architecture. Community features that do not span integration boundaries are more likely to work as expected.
- Features primarily used when targeting hosted environments such as Linux or BSD, might have significant limitations, or might not be applicable, when targeting bare-metal environments.
- The Clang implementations of compiler features, particularly those that have been present for a long time in other toolchains, are likely to be mature. The functionality of new features, such as support for new language features, is likely to be less mature and therefore more likely to have limited functionality.

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.
- C++ static initialization of local variables is not thread-safe when linked against the standard C++ libraries. For thread-safety, you must provide your own implementation of thread-safe functions as described in *Standard C++ library implementation definition*.

  Note

  This restriction does not apply to the [ALPHA]-supported multi-threaded C++ libraries. Contact the ARM Support team for more details.

- Use of C11 library features is unsupported.
- Any community feature that exclusively pertains to non-ARM architectures is not supported by ARM Compiler 6.
- Compilation for targets that implement architectures older that ARMv7 or ARMv6-M is not supported.

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 [ALPHA] and [COMMUNITY] features. See *Support level definitions* on page 1-13.

Base LLVM components

ARM Compiler 6 is based on the following LLVM components:

Table 1-1 LLVM component versions

Component	Version	More information
Clang	3.9	*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 and therefore these are community features. 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.

Table 1-2 Language support levels (continued)

Language standard	Support level
C++11	Supported, with the following exceptions: [ALPHA] Concurrency constructs available through the following standard library headers are [ALPHA] supported: • `<thread>` • `<mutex>` • `<shared_mutex>` • `<condition_variable>` • `<future>` • `<chrono>` • `<atomic>` • For more details, contact the ARM Support team. 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 and therefore these are community features.

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.

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`.
`-march=list`	Displays a list of all the supported architectures for your target.
`-mcpu=`*`name`*	Generates code for the specified processor, for example `-mcpu=cortex-a53`, `-mcpu=cortex-a57`, or `-mcpu=cortex-a15`.
`-mcpu=list`	Displays a list of all the supported processors for your target.
`-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.

Common `armlink` options include the following:

Table 1-4 armlink common options

Option	Description
`--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:

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.

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\n");
    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.

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.
`-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`.

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

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`

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.
2.4 Options for controlling diagnostics with the other tools on page 2-28.

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.

`-W`*`foo`*
: Enable warning *`foo`*.

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

`-w`
: Suppress all warnings.

`-Weverything`
: Enable all warnings.

`-Wpedantic`
: Generate warnings if code violates strict ISO C and ISO C++.

`-pedantic`
: Generate warnings if code violates strict ISO C and ISO C++.

`-pedantic-errors`
: Generate errors if code violates strict ISO C and ISO C++.

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 7-52.

Related information

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

2.3 Pragmas for controlling diagnostics with armclang

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

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

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

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

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

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

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

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

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

The compiler provides appropriate diagnostic names in the diagnostic output.

Note

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

Related information

-W.

2.4 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
```

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

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. Specify the execution state using the `--target` option.

   The execution state can be AArch64 or AArch32 depending on the processor.
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) For processors that support both ARM and Thumb, 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.

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.

You can specify feature modifiers with `-mcpu` and `-march`. 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 the cryptographic extensions for AArch64 targets use the `-mcpu=`*name*`+nocrypto` option.

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`
ARMv7-A	Generic	A32	`armclang --target=arm-arm-none-eabi -march=armv7-a test.c`

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

Architecture	Processor	Instruction set	armclang command
ARMv7-A	Cortex-A9	A32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-r7 test.c`
ARMv7-A	Cortex-A15	T32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-r7 -mthumb test.c`
ARMv7-R	Cortex-R7	A32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-r7 test.c`
ARMv7-R	Cortex-R7	T32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-r7 -mthumb test.c`
ARMv7-M	Generic	T32	`armclang --target=arm-arm-none-eabi -march=armv7-m test.c`
ARMv6-M	Cortex-M0	T32	`armclang --target=arm-arm-none-eabi -mcpu=cortex-m0 test.c`
ARMv8-M.Mainline	Generic	T32	`armclang --target=arm-arm-none-eabi -march=armv8-m.main test.c`
ARMv8-M.Baseline	Generic	T32	`armclang --target=arm-arm-none-eabi -march=armv8-m.base test.c`

Related information

-mcpu.
--target.
-marm.
-mthumb.
-mfpu.

3.2 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.

3.3 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.

3.4 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.
About floating-point support.

3.5 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.

Note

- Bare-metal PIE support is deprecated.
- There is support for `-fropi` and `-frwpi` in `armclang`. You can use these options to create bare-metal position independent executables.

Position independent code uses PC-relative addressing modes where possible and otherwise accesses global data via the *Global Offset Table* (GOT). The address entries in the GOT and initialized pointers in the data area are updated with the executable load address when the executable runs for the first time.

All objects and libraries 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_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_relocation_model=pic --bare_metal_pie hello.bc -o hello
```

Restrictions

A bare-metal PIE executable must conform to the following:

- AArch32 state only.
- The `.got` section must be placed in a writable region.
- All references to symbols must be resolved at link time.
- The image must be linked Position Independent with a base address of 0x0.
- The code and data must be linked at a fixed offset from each other.

- The stack must be set up before the runtime relocation routine `__arm_relocate_pie_` is called. This means that the stack initialization code must only use PC-relative addressing if it is part of the image code.
- It is the responsibility of the target platform that loads the PIE to ensure that the ZI region is zero-initialized.
- When writing assembly code for position independence, 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.
-fropi.
-frwpi.

3.6 Execute-only memory

Execute-only memory (XOM) allows only instruction fetches. Read and write accesses are not allowed.

Execute-only memory allows you to protect your intellectual property by preventing executable code being read by users. For example, you can place firmware in execute-only memory and load user code and drivers separately. Placing the firmware in execute-only memory prevents users from trivially reading the code.

Note

The ARM architecture does not directly support execute-only memory. Execute-only memory is supported at the memory device level.

3.7 Building applications for execute-only memory

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

Note

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

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

Procedure

1. Compile your C or C++ code using the `-mexecute-only` option.
   ```
   armclang --target=arm-arm-none-eabi -march=armv7-m -mexecute-only -c test.c -o test.o
   ```
   The `-mexecute-only` option prevents the compiler from generating any data accesses to the code sections.

   To keep code and data in separate sections, the compiler disables the placement of literal pools inline with code.

   Compiled execute-only code sections in the ELF object file are marked with the `SHF_ARM_NOREAD` flag.
2. Specify the memory map to the linker using either of the following:
   - The `+XO` selector in a scatter file.
   - The `armlink --xo-base` option on the command-line.

   ```
   armlink --xo-base=0x8000 test.o -o test.axf
   ```
   The XO execution region is placed in a separate load region from the RO, RW, and ZI execution regions.

   Note

   If you do not specify `--xo-base`, then by default:
   - The XO execution region is placed immediately before the RO execution region, at address `0x8000`.
   - All execution regions are in the same load region.

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

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
```

Related information

GNU Binutils - Using as.

Migrating ARM syntax assembly code to GNU syntax.

4.2 Preprocessing assembly code

The C preprocessor must resolve assembly code that contains C directives, for example `#include` or `#define`, before assembling.

By default, `armclang` uses the assembly code source file suffix to determine whether to run the C preprocessor:

- The `.s` (lowercase) suffix indicates assembly code that does not require preprocessing.
- The `.S` (uppercase) suffix indicates assembly code that requires preprocessing.

The `-x` option lets you override the default by specifying the language of the subsequent source files, rather than inferring the language from the file suffix. Specifically, `-x assembler-with-cpp` indicates that the assembly code contains C directives and `armclang` must run the C preprocessor. The `-x` option only applies to input files that follow it on the command line.

Note

Do not confuse the `.ifdef` assembler directive with the preprocessor `#ifdef` directive:

- The preprocessor `#ifdef` directive checks for the presence of preprocessor macros, These macros are defined using the `#define` preprocessor directive or the `armclang -D` command-line option.
- The `armclang` integrated assembler `.ifdef` directive checks for code symbols. These symbols are defined using labels or the `.set` directive.

The preprocessor runs first and performs textual substitutions on the source code. This stage is when the `#ifdef` directive is processed. The source code is then passed onto the assembler, when the `.ifdef` directive is processed.

To preprocess an assembly code source file, do one of the following:

- Ensure that the assembly code filename has a `.S` suffix.

  For example:

  ```
  armclang --target=arm-arm-none-eabi -march=armv8-a -E test.S
  ```

- Use the `-x assembler-with-cpp` option to tell `armclang` that the assembly source file requires preprocessing. This option is useful when you have existing source files with the lowercase extension `.s`.

  For example:

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

Note

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

Related information

Command-line options for preprocessing assembly source code.
-E armclang option.
-x armclang option.

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 section:

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
```

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

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. This option balances code size against performance.

`-Oz`
: Optimizes for smaller code size.

Note

You can also set the optimization level with the `armlink` option `--lto_level`. The levels correspond to the `armclang` optimization levels.

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.

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 files containing bitcode together and sends them to the link time optimizer (`libLTO`). 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.

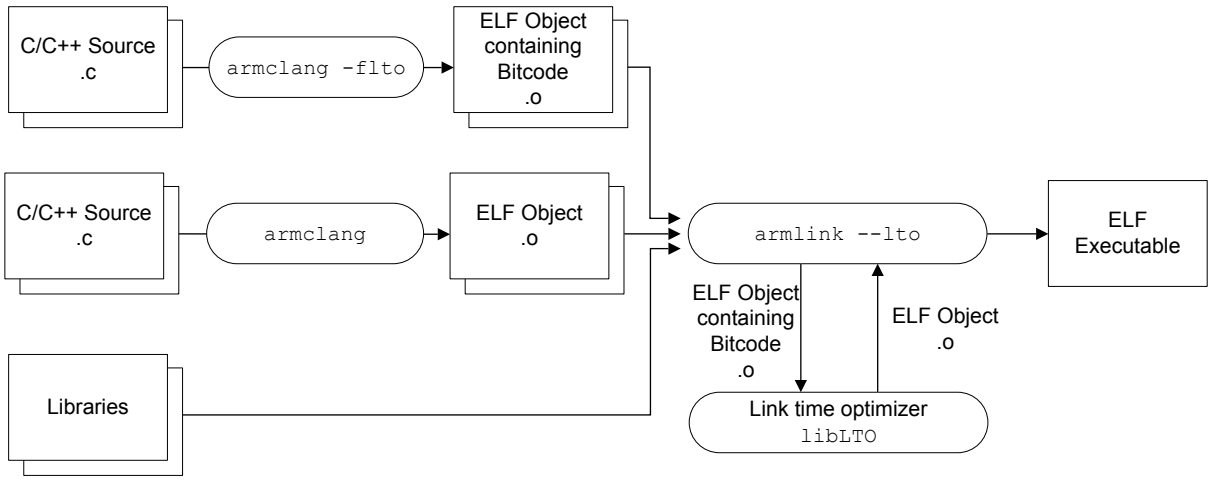


Figure 6-1 Link time optimization

Note

In this figure, ELF Object containing Bitcode is an ELF file that does not contain normal code and data. Instead, it contains a section called `.llvmbc` that holds LLVM bitcode.

Section `.llvmbc` is reserved. You must not create an `.llvmbc` section with, for example `__attribute__((section(".llvmbc")))`.

Caution

Link Time Optimization performs aggressive optimizations. Sometimes this can result in large chunks of code being removed.

This section contains the following subsections:

- *6.2.1 Enabling link time optimization* on page 6-48.
- *6.2.2 Restrictions with link time optimization* on page 6-49.

6.2.1 Enabling link time optimization

You must enable link time optimization in both `armclang` and `armlink`.

To enable link time optimization:

1. At compilation time, use the `armclang` option `-flto` to produce ELF files suitable for link time optimization. These ELF files contain bitcode in a `.llvmbc` section.
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 ELF files `src1.o`, `src2.o`, and `src3.o`. These ELF files contain bitcode.
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.o
armclang --target=arm-arm-none-eabi -march=armv8-a -c -flto src3.c -o src3.o
armlink --lto src1.o src2.o src3.o -o output.axf
```

This example does the following:

1. `armclang` compiles the C source file `src1.c` to the ELF object file `src1.o`.
2. `armclang` compiles the C source files `src2.c` and `src3.c` to the ELF files `src2.o` and `src3.o`. These ELF files contain bitcode.
3. `armlink` passes the bitcode files `src2.o` and `src3.o` to the link time optimizer to produce a single optimized ELF object file.
4. `armlink` combines the ELF object file `src1.o` with the object file produced by the link time optimizer to create the executable `output.axf`.

Related references

6.2.2 Restrictions with link time optimization on page 6-49.

Related information

-flto.

--lto armlink option.

--keep armlink option.

6.2.2 Restrictions with link time optimization

Link time optimization has a few restrictions in ARM Compiler 6. Future releases might have fewer restrictions and more features. The user interface to link time optimization might change in future releases.

No bitcode libraries

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

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

Partial linking

The `armlink` option `--partial` only works with ELF files. The linker gives an error message if it detects a file containing bitcode.

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 are matched by input section selectors as normal.

Use the `armlink` option `--lto_intermediate_filename` to name the ELF object file output. You can reference this ELF file name in the scatter file.

ARM recommends that link time optimization is only performed on code and data that does not require precise placement in the scatter file, with general input section selectors such as `*(+RO)` and `.ANY(+RO)` used to select sections generated by link time optimization.
It is not possible to match bitcode in `.llvmbc` sections by name in a scatter file.

Note

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

Executable and library compatibility

The `armclang` executable and the `libLTO` library must come from the same ARM Compiler 6 installation. Any use of `libLTO` other than that supplied with ARM Compiler 6 is unsupported.

Other restrictions

- You cannot currently use link time optimization for building ROPI/RWPI images.
- Object files that are produced by the link time optimization contain build attributes that are the default for the target architecture. If you use the `armlink` options `--cpu` or `--fpu` when link time optimization is enabled, `armlink` can incorrectly report that the attributes in the file produced by the link time optimizer are incompatible with the provided attributes.
- Link Time Optimization does not honor `armclang` options `-ffunction-sections` and `-fdata-sections`.
- Link Time Optimization does not honor the `armclang -mexecute-only` option. If you use the `armclang -flto` or `-Omax` options, then the compiler cannot generate execute-only code.

Related references

6.2.1 Enabling link time optimization on page 6-48.

Related information

-flto.

--lto armlink option.

--keep armlink option.

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.

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

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	Decrementing loop

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

The following table shows the corresponding disassembly 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	Decrementing loop

`fact1:`
`  mov  r1, r0`
`  mov  r0, #1`
`  cmp  r1, #1`
`  bxlt  lr`
`  mov  r2, #0`
`.LBB0_1:`
`  add  r2, r2, #1`
`  mul  r0, r0, r2`
`  cmp  r1, r2`
`  bne  .LBB0_1`
`  bx  lr`
	`fact2:`
	`  mov  r1, r0`
	`  mov  r0, #1`
	`  cmp  r1, #0`
	`  bxeq  lr`
	`.LBB1_1:`
	`  mul  r0, r0, r1`
	`  subs  r1, r1, #1`
	`  bne  .LBB1_1`
	`  bx  lr`

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

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(); ...);
```

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.

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`. 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`.

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

Bit-counting loop	Unrolled bit-counting loop

`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`
	`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`

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

7.3 Effect of the volatile keyword on compiler optimization

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

What volatile means

The declaration of a variable as volatile tells the compiler that the variable can be modified at any time externally to the implementation, for example:

- By the operating system.
- By another thread of execution such as an interrupt routine or signal handler.
- By hardware.

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

When to use volatile

Use the volatile keyword for variables that might be modified external to the implementation.

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

Another example is that a variable might be used to implement a sleep or timer delay. If the variable appears unused, the compiler might remove the timer delay code, unless the variable is declared as volatile.

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

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

Potential problems when not using volatile

When a volatile variable is not declared as volatile, the compiler assumes that its value cannot be modified externally to the implementation. Therefore, the compiler might perform unwanted optimizations. 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.
- Optimization might result in the removal of code that implements deliberate timing delays.

Example of infinite loop when not using the volatile keyword

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

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

Nonvolatile version of buffer loop	Volatile version of buffer loop

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

Both of these routines increment a counter in a loop until a status flag `buffer_full` is set to true. The state of `buffer_full` can change asynchronously with program flow.

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

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

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

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

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. Compile with the `-g` option to generate the necessary DWARF information.
- 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.

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.

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.

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.

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);
```

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.

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.

Chapter 8
Mapping code and data to target memory

Describes how to place your code and data into the correct areas of memory on your target hardware.

It contains the following sections:

8.1 Overlay support in ARM® Compiler

There are situations when you might want to load some code in memory, then replace it with different code. For example, your system might have memory constraints that mean you cannot load all code into memory at the same time.

The solution is to create an overlay region where each piece of overlaid code is unloaded and loaded by an overlay manager. ARM Compiler supports:

- An automatic overlay mechanism, where the linker decides how your code sections get allocated to overlay regions.
- A manual overlay mechanism, where you manually arrange the allocation of the code sections.

Related concepts

8.2 Automatic overlay support on page 8-68.

8.3 Manual overlay support on page 8-73.

Related information

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

AREA.

Execution region attributes.

--emit_debug_overlay_section linker option.

--overlay_veneers linker option.

8.2 Automatic overlay support

For the linker to automatically allocate code sections to overlay regions, you must modify your C or assembly code to identify the parts to be overlaid. You must also set up a scatter file to locate the overlays.

The automatic overlay mechanism consists of:

- Special section names that you can use in your object files to mark code as overlaid.
- The `AUTO_OVERLAY` execution region attribute. Use this in a scatter file to indicate regions of memory where the linker assigns the overlay sections for loading into at runtime.
- The command-line option `--overlay-veneers` to make the linker redirect calls between overlays to a veneer that lets an overlay manager unload and load the correct overlays.
- A set of data tables and symbol names provided by the linker that you can use to write the overlay manager.
- The `armlink --emit_debug_overlay_section` command-line options to add extra debug information to the image. This option permits an overlay-aware debugger to track which overlay is currently active.

This section contains the following subsections:

8.2.1 Automatically placing code sections in overlay regions

ARM Compiler can automatically place code sections into overlay regions.

You identify the sections in your code that are to become overlays by giving them names of the form `.ARM.overlay`*N*, where *N* is an integer identifier. You then use a scatter file to indicate those regions of memory where `armlink` is to assign the overlays for loading at runtime.

Each overlay region corresponds to an execution region that has the attribute `AUTO_OVERLAY` assigned in the scatter file. `armlink` allocates one set of integer identifiers to each of these overlay regions. It allocates another set of integer identifiers to each overlaid section with the name `.ARM.overlayN` that is defined in the object files.

Note

The numbers assigned to the overlay sections in your object files do not match up to the numbers that you put in the `.ARM.overlay`*N* section names.

Procedure

1. Declare the functions that you want the `armlink` automatic overlay mechanism to process.
   - In C, use a function attribute, for example:

     ```
     __attribute__((section(".ARM.overlay1"))) void foo(void) { ... }

     __attribute__((section(".ARM.overlay2"))) void bar(void) { ... }
     ```

   - In the `armclang` integrated assembler syntax, use the `.section` directive, for example:

     ```
         .section    .ARM.overlay1,"ax",%progbits
         .globl    foo
         .p2align    2
         .type    foo,%function
     foo:                                    @ @foo
         ...
         .fnend

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

```
    .p2align    2
    .type    bar,%function
bar:                                    @ @bar
    ...
    .fnend
```

- In ARM syntax assembly, use the `AREA` directive, for example:

```
    AREA |.ARM.overlay1|,CODE
foo PROC
    ...
    ENDP

    AREA |.ARM.overlay2|,CODE
bar PROC
    ...
    ENDP
```

Note

You can choose to overlay or not overlay code sections. Data sections must never be overlaid.

2. Specify the locations to load the code sections from and to in a scatter file. Use the `AUTO_OVERLAY` keyword on one or more execution regions.

   The execution regions must not have any section selectors. For example:

```
OVERLAY_LOAD_REGION 0x10000000
{
    OVERLAY_EXECUTE_REGION_A 0x20000000 AUTO_OVERLAY 0x10000 { }
    OVERLAY_EXECUTE_REGION_B 0x20010000 AUTO_OVERLAY 0x10000 { }
}
```

   In this example, `armlink` emits a program header table entry that loads all the overlay data starting at address `0x10000000`. Also, each overlay is relocated so that it runs correctly if copied to address `0x20000000` or `0x20010000`. `armlink` chooses one of these addresses for each overlay.

3. When linking, specify the `--overlay_veneers` command-line option. This option causes `armlink` to arrange function calls between two overlays, or between non-overlaid code and an overlay, to be diverted through the entry point of an overlay manager.

   To permit an overlay-aware debugger to track the overlay that is active, specify the `armlink --emit_debug_overlay_section` command-line option.

Related information

__attribute__((section("name"))) function attribute.
AREA.
Execution region attributes.
--emit_debug_overlay_section linker option.
--overlay_veneers linker option.

8.2.2 Overlay veneer

`armlink` can generate an overlay veneer for each function call between two overlays, or between non-overlaid code and an overlay.

A function call or return can transfer control between two overlays or between non-overlaid code and an overlay. If the target function is not already present at its intended execution address, then the target overlay has to be loaded.

To detect whether the target overlay is present, `armlink` can arrange for all such function calls to be diverted through the overlay manager entry point, `__ARM_overlay_entry`. To enable this feature, use the `armlink` command-line option `--overlay_veneers`. This option causes a veneer to be generated for each affected function call, so that the call instruction, typically a `BL` instruction, points at the veneer instead of the target function. The veneer in turn saves some registers on the stack, loads some

information about the target function and the overlay that it is in, and transfers control to the overlay manager entry point. The overlay manager must then:

- Ensure that the correct overlay is loaded and then transfer control to the target function.
- Restore the stack and registers to the state they were left in by the original `BL` instruction.
- If the function call originated inside an overlay, make sure that returning from the called function reloads the overlay being returned to.

Related information

--overlay_veneers linker option.

8.2.3 Overlay data tables

`armlink` provides various symbols that point to a piece of read-only data, mostly arrays. This data describes the collection of overlays and overlay regions in the image.

The symbols are:

`Region$$Table$$AutoOverlay`
: This symbol points to an array containing two 32-bit pointers per overlay region. For each region, the two pointers give the start address and end address of the overlay region. The start address is the first byte in the region. The end address is the first byte beyond the end of the region. The overlay manager can use this symbol to identify when the return address of a calling function is in an overlay region. In this case, a return thunk might be required.

Note

The regions are always sorted in ascending order of start address.

`Region$$Count$$AutoOverlay`
: This symbol points to a single 16-bit integer (an unsigned short) giving the total number of overlay regions. That is, the number of entries in the arrays `Region$$Table$$AutoOverlay` and `CurrLoad$$Table$$AutoOverlay`.

`Overlay$$Map$$AutoOverlay`
: This symbol points to an array containing a 16-bit integer (an unsigned short) per overlay. For each overlay, this table indicates which overlay region the overlay expects to be loaded into to run correctly.

`Size$$Table$$AutoOverlay`
: This symbol points to an array containing a 32-bit word per overlay. For each overlay, this table gives the exact size of the data for the overlay. This size might be less than the size of its containing overlay region, because overlays typically do not fill their regions exactly.

In addition to the read-only tables, `armlink` also provides one piece of read/write memory:

`CurrLoad$$Table$$AutoOverlay`
: This symbol points to an array containing a 16-bit integer (an unsigned short) for each overlay region. The array is intended for the overlay manager to store the identifier of the currently loaded overlay in each region. The overlay manager can then avoid reloading an already-loaded overlay.

All these data tables are optional. If your code does not refer to any particular table, then it is omitted from the image.

Related concepts

8.2 Automatic overlay support on page 8-68.

8.2.4 Limitations of automatic overlay support

There are some limitations when using the automatic overlay feature.

The following limitations apply:

- The automatic overlay feature does not support C++.
- If you assign multiple functions to the same named section `.ARM.overlay`*N*, then `armlink` treats them as different overlays. `armlink` assigns a different integer ID to each overlay.
- The `armlink --any_placement` command-line option is currently ignored for the automatic overlay sections.
- The overlay system automatically generates veneers for direct calls between overlays, and between non-overlaid code and overlaid code. It automatically arranges that indirect calls through function pointers to functions in overlays work. However, there is one type of indirect function call that is not correctly fixed up, namely the case where you take a pointer to a non-overlaid function and pass that pointer into an overlay that calls it. In that situation, `armlink` has no way to insert a call to the overlay veneer. Therefore, the overlay manager has no opportunity to arrange to reload the overlay on behalf of the calling function on return.

  In simple cases, this can still work. However, if the non-overlaid function calls something in a second overlay that conflicts with the overlay of its calling function, then a runtime failure occurs. For example:

```
__attribute__((section(".ARM.overlay1"))) void innermost(void)
{
    // do something
}

void non_overlaid(void)
{
    innermost();
}

typedef void (*function_pointer)(void);

__attribute__((section(“.ARM.overlay2”))) void call_via_ptr(function_pointer f)
{
    f();
}

int main(void)
{
    // Call the overlaid function call_via_ptr() and pass it a pointer
    // to non_overlaid(). non_overlaid() then calls the function
    // innermost() in another overlay. If call_via_ptr() and innermost()
    // are allocated to the same overlay region by the linker, then there
    // is no way for call_via_ptr to have been reloaded by the time control
    // has to return to it from non_overlaid().

    call_via_ptr(non_overlaid);
}
```

Related concepts

8.2 Automatic overlay support on page 8-68.

8.2.5 Writing an overlay manager for automatically placed overlays

To write an overlay manager to handle loading and unloading of overlays, you must provide an implementation of the overlay manager entry point.

The overlay manager entry point `__ARM_overlay_entry` is the location that the linker-generated veneers expect to jump to. The linker also provides some tables of data to enable the overlay manager to find the overlays and the overlay regions to load.

The entry point is called by the linker overlay veneers as follows:

- r0 contains the integer identifier of the overlay containing the target function.
- r1 contains the execution address of the target function. That is, the address that the function appears at when its overlay is loaded.
- The overlay veneer pushes six 32-bit words onto the stack. These words comprise the values of the r0, r1, r2, r3, r12, and lr registers of the calling function. If the call instruction is a `BL`, the value of lr is the one written into lr by the `BL` instruction, not the one before the `BL`.

The overlay manager has to:

1. Load the target overlay.
2. Restore all six of the registers from the stack.
3. Transfer control to the address of the target function that is passed in r1.

The overlay manager might also have to modify the value it passes to the calling function in lr to point at a return thunk routine. This routine would reload the overlay of the calling function and then return control to the original value of the lr of the calling function.

There is no sensible place already available to store the original value of lr for the return thunk to use. For example, there is nowhere on the stack that can contain the value. Therefore, the overlay manager has to maintain its own stack-organized data structure. The data structure contains the saved lr value and the corresponding overlay ID for each time the overlay manager substitutes a return thunk during a function call, and keeps it synchronized with the main call stack.

Note

Because this extra parallel stack has to be maintained, then you cannot use stack manipulations such as cooperative or preemptive thread switching, coroutines, and setjmp/longjmp, unless it is customized to keep the parallel stack of the overlay manager consistent.

The `armlink --info=auto_overlays` option causes the linker to write out a text summary of the overlays in the image it outputs. The summary consists of the integer ID, start address, and size of each overlay. You can use this information to extract the overlays from the image, perhaps from the `fromelf --bin` output. You can then put them in a separate peripheral storage system. Therefore, you still know which chunk of data goes with which overlay ID when you have to load one of them in the overlay manager.

Related concepts

8.2 Automatic overlay support on page 8-68.

Related information

__attribute__((section("name"))) function attribute.
AREA.
Execution region attributes.
--emit_debug_overlay_section linker option.
--overlay_veneers linker option.

8.3 Manual overlay support

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

The manual overlay mechanism consists of:

- The `OVERLAY` attribute for load regions and execution regions. Use this attribute in a scatter file to indicate regions of memory where the linker assigns the overlay sections for loading into at runtime.
- The following `armlink` command-line options to add extra debug information to the image:
  - `--emit_debug_overlay_relocs`.
  - `--emit_debug_overlay_section`.

  This extra debug information permits an overlay-aware debugger to track which overlay is active.

This section contains the following subsections:

- *8.3.1 Manually placing code sections in overlay regions* on page 8-73.
- *8.3.2 Writing an overlay manager for manually placed overlays* on page 8-74.

8.3.1 Manually placing code sections in overlay regions

You can place multiple execution regions at the same address with overlays.

The `OVERLAY` attribute allows you to place multiple execution regions at the same address. An overlay manager is required to make sure that only one execution region is instantiated at a time. ARM Compiler does not provide an overlay manager.

The following example shows the definition of a static section in RAM followed by a series of overlays. Here, only one of these sections is instantiated at a time.

```
EMB_APP 0x8000
{
    …
    STATIC_RAM 0x0                  ; contains most of the RW and ZI code/data
    {
            * (+RW,+ZI)
    }
    OVERLAY_A_RAM 0x1000 OVERLAY    ; start address of overlay…
    {
            module1.o (+RW,+ZI)
    }
    OVERLAY_B_RAM 0x1000 OVERLAY
    {
            module2.o (+RW,+ZI)
    }
    …                         ; rest of scatter-loading description
}
```

The C library at startup does not initialize a region that is marked as `OVERLAY`. The contents of the memory that is used by the overlay region is the responsibility of an overlay manager. If the region contains initialized data, use the `NOCOMPRESS` attribute to prevent RW data compression.

You can use the linker defined symbols to obtain the addresses that are required to copy the code and data.

You can use the `OVERLAY` attribute on a single region that is not at the same address as a different region. Therefore, you can use an overlay region as a method to prevent the initialization of particular regions by the C library startup code. As with any overlay region, you must manually initialize them in your code.

An overlay region can have a relative base. The behavior of an overlay region with a `+offset` base address depends on the regions that precede it and the value of `+offset`. If they have the same `+offset` value, the linker places consecutive `+offset` regions at the same base address.

When a `+offset` execution region ER follows a contiguous overlapping block of overlay execution regions the base address of ER is:

`limit address of the overlapping block of overlay execution regions + offset`

The following table shows the effect of `+offset` when used with the `OVERLAY` attribute. `REGION1` appears immediately before `REGION2` in the scatter file:

Table 8-1 Using relative offset in overlays

REGION1 is set with `OVERLAY`	`+offset`	REGION2 Base Address
NO	*<offset>*	`REGION1 Limit +` *<offset>*
YES	`+0`	`REGION1 Base Address`
YES	*<non-zero offset>*	`REGION1 Limit +` *<non-zero offset>*

The following example shows the use of relative offsets with overlays and the effect on execution region addresses:

```
EMB_APP 0x8000
{
    CODE 0x8000
    {
        *(+RO)
    }
    # REGION1 Base = CODE limit
    REGION1 +0 OVERLAY
    {
        module1.o(*)
    }
    # REGION2 Base = REGION1 Base
    REGION2 +0 OVERLAY
    {
        module2.o(*)
    }
    # REGION3 Base = REGION2 Base = REGION1 Base
    REGION3 +0 OVERLAY
    {
        module3.o(*)
    }
    # REGION4 Base = REGION3 Limit + 4
    Region4 +4 OVERLAY
    {
        module4.o(*)
    }
}
```

If the length of the non-overlay area is unknown, you can use a zero relative offset to specify the start address of an overlay so that it is placed immediately after the end of the static section.

Related information

Load region descriptions.
Load region attributes.
Inheritance rules for load region address attributes.
Considerations when using a relative address +offset for a load region.
Considerations when using a relative address +offset for execution regions.
--emit_debug_overlay_relocs linker option.
--emit_debug_overlay_section linker option.
ABI for the ARM Architecture: Support for Debugging Overlaid Programs.

8.3.2 Writing an overlay manager for manually placed overlays

Overlays are not automatically copied to their runtime location when a function within the overlay is called. Therefore, you must write an overlay manager to copy overlays.

The overlay manager copies the required overlay to its execution address, and records the overlay that is in use at any one time. The overlay manager runs throughout the application, and is called whenever overlay loading is required. For instance, the overlay manager can be called before every function call that might require a different overlay segment to be loaded.

The overlay manager must ensure that the correct overlay segment is loaded before calling any function in that segment. If a function from one overlay is called while a different overlay is loaded, then some kind of runtime failure occurs. If such a failure is a possibility, the linker and compiler do not warn you because it is not statically determinable. The same is true for a data overlay.

The central component of this overlay manager is a routine to copy code and data from the load address to the execution address. This routine is based around the following linker defined symbols:

- `Load$$execution_region_name$$Base`, the load address.
- `Image$$execution_region_name$$Base`, the execution address.
- `Image$$execution_region_name$$Length`, the length of the execution region.

The implementation of the overlay manager depends on the system requirements. This procedure shows a simple method of implementing an overlay manager. The downloadable example contains a `Readme.txt` file that describes details of each source file.

The copy routine that is called `load_overlay()` is implemented in `overlay_manager.c`. The routine uses `memcpy()` and `memset()` functions to copy CODE and RW data overlays, and to clear ZI data overlays.

Note

For RW data overlays, it is necessary to disable RW data compression for the whole project. You can disable compression with the linker command-line option `--datacompressor off`, or you can mark the execution region with the attribute `NOCOMPRESS`.

The assembly file `overlay_list.s` lists all the required symbols. This file defines and exports two common base addresses and a RAM space that is mapped to the overlay structure table:

```
code_base
data_base
overlay_regions
```

As specified in the scatter file, the two functions, `func1()` and `func2()`, and their corresponding data are placed in `CODE_ONE`, `CODE_TWO`, `DATA_ONE`, `DATA_TWO` regions, respectively. `armlink` has a special mechanism for replacing calls to functions with stubs. To use this mechanism, write a small stub for each function in the overlay that might be called from outside the overlay.

In this example, two stub functions `$Sub$$func1()` and `$Sub$$func2()` are created for the two functions `func1()` and `func2()` in `overlay_stubs.c`. These stubs call the overlay-loading function `load_overlay()` to load the corresponding overlay. After the overlay manager finishes its overlay loading task, the stub function can then call `$Super$$func1` to call the loaded function `func1()` in the overlay.

Procedure

1. Create the `overlay_manager.c` program to copy the correct overlay to the runtime addresses.

```
// overlay_manager.c
/* Basic overlay manager */

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

/* Number of overlays present */
#define NUM_OVERLAYS 2

/* struct to hold addresses and lengths */
typedef struct overlay_region_t_struct
{
    void* load_ro_base;
    void* load_rw_base;
    void* exec_zi_base;
    unsigned int ro_length;
    unsigned int zi_length;
} overlay_region_t;

/* Record for current overlay */
```

```
int current_overlay = 0;

/* Array describing the overlays */
extern const overlay_region_t overlay_regions[NUM_OVERLAYS];

/* execution bases of the overlay regions - defined in overlay_list.s */
extern void * const code_base;
extern void * const data_base;

void load_overlay(int n)
{
    const overlay_region_t * selected_region;

    if(n == current_overlay)
    {
        printf("Overlay %d already loaded.\n", n);
        return;
    }

    /* boundary check */
    if(n<1 || n>NUM_OVERLAYS)
    {
        printf("Error - invalid overlay number %d specified\n", n);
        exit(1);
    }

    /* Load the corresponding overlay */
    printf("Loading overlay %d...\n", n);

    /* set selected region */
    selected_region = &overlay_regions[n-1];

    /* load code overlay */
    memcpy(code_base, selected_region->load_ro_base, selected_region->ro_length);

    /* load data overlay */
    memcpy(data_base, selected_region->load_rw_base,
           (unsigned int)selected_region->exec_zi_base - (unsigned int)data_base);

    /* Comment out the next line if your overlays have any static ZI variables
     * and should not be reinitialized each time, and move them out of the
     * overlay region in your scatter file */
    memset(selected_region->exec_zi_base, 0, selected_region->zi_length);

    /* update record of current overlay */
    current_overlay=n;

    printf("...Done.\n");

}
```

2. Create a separate source file for each of the functions `func1()` and `func2()`.

```
// func1.c
#include <stdio.h>
#include <stdlib.h>

extern void foo(int x);

// Some RW and ZI data
char* func1_string = "func1 called\n";
int func1_values[20];

void func1(void)
{
    unsigned int i;
    printf("%s\n", func1_string);
    for(i = 19; i; i--)
    {
        func1_values[i] = rand();
        foo(i);
        printf("%d ", func1_values[i]);
    }
    printf("\n");
}
```

```
// func2.c
#include <stdio.h>

extern void foo(int x);

// Some RW and ZI data
char* func2_string = "func2 called\n";
int func2_values[10];
```

```
void func2(void)
{
    printf("%s\n", func2_string);
    foo(func2_values[9]);
}
```

3. Create the `main.c` program to demonstrate the overlay mechanism.

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


/* Functions provided by the overlays */
extern void func1(void);
extern void func2(void);

int main(void)
{
    printf("Start of main()...\n");
    func1();
    func2();

    /*
     * Call func2() again to demonstrate that we don't need to
     * reload the overlay
     */
    func2();

    func1();
    printf("End of main()...\n");

    return 0;
}

void foo(int x)
{
    return;
}
```

4. Create `overlay_stubs.c` to provide two stub functions `$Sub$$func1()` and `$Sub$$func2()` for the two functions `func1()` and `func2()`.

```
// overlay_stub.c
extern void $Super$$func1(void);
extern void $Super$$func2(void);

extern void load_overlay(int n);

void $Sub$$func1(void)
{
    load_overlay(1);
    $Super$$func1();
}

void $Sub$$func2(void)
{
    load_overlay(2);
    $Super$$func2();
}
```

5. Create `overlay_list.s` that lists all the required symbols.

```
; overlay_list.s
    AREA    overlay_list, DATA, READONLY

    ; Linker-defined symbols to use

    IMPORT ||Load$$CODE_ONE$$Base||
    IMPORT ||Load$$CODE_TWO$$Base||
    IMPORT ||Load$$DATA_ONE$$Base||
    IMPORT ||Load$$DATA_TWO$$Base||

    IMPORT ||Image$$CODE_ONE$$Base||
    IMPORT ||Image$$DATA_ONE$$Base||
    IMPORT ||Image$$DATA_ONE$$ZI$$Base||
    IMPORT ||Image$$DATA_TWO$$ZI$$Base||

    IMPORT ||Image$$CODE_ONE$$Length||
    IMPORT ||Image$$CODE_TWO$$Length||

    IMPORT ||Image$$DATA_ONE$$ZI$$Length||
    IMPORT ||Image$$DATA_TWO$$ZI$$Length||
```

```
    ; Symbols to export

    EXPORT code_base
    EXPORT data_base
    EXPORT overlay_regions

; Common base execution addresses of the two OVERLAY regions

code_base DCD ||Image$$CODE_ONE$$Base||
data_base DCD ||Image$$DATA_ONE$$Base||

; Array of details for each region -
; see overlay_manager.c for structure layout

overlay_regions
; overlay 1
    DCD ||Load$$CODE_ONE$$Base||
    DCD ||Load$$DATA_ONE$$Base||
    DCD ||Image$$DATA_ONE$$ZI$$Base||
    DCD ||Image$$CODE_ONE$$Length||
    DCD ||Image$$DATA_ONE$$ZI$$Length||

; overlay 2
    DCD ||Load$$CODE_TWO$$Base||
    DCD ||Load$$DATA_TWO$$Base||
    DCD ||Image$$DATA_TWO$$ZI$$Base||
    DCD ||Image$$CODE_TWO$$Length||
    DCD ||Image$$DATA_TWO$$ZI$$Length||

    END
```

6. Create `retarget.c` to retarget the `__user_initial_stackheap` function.

```
// retarget.c
#include <rt_misc.h>

extern unsigned int Image$$HEAP$$ZI$$Base;
extern unsigned int Image$$STACKS$$ZI$$Limit;

__value_in_regs struct __initial_stackheap __user_initial_stackheap(
        unsigned R0, unsigned SP, unsigned R2, unsigned SL)
{
    struct __initial_stackheap config;

    config.heap_base = (unsigned int)&Image$$HEAP$$ZI$$Base;
    config.stack_base = (unsigned int)&Image$$STACKS$$ZI$$Limit;

    return config;
}
```

7. Create the scatter file, `embedded_scat.scat`.

```
; embedded_scat.scat
;;; Copyright ARM Ltd 2002. All rights reserved.

;; Embedded scatter file

ROM_LOAD 0x24000000 0x04000000
{
    ROM_EXEC 0x24000000 0x04000000
    {
        * (InRoot$$Sections)      ; All library sections that must be in a root region
                                  ; e.g. __main.o, __scatter*.o, * (Region$$Table)
        * (+RO)                   ; All other code
    }

    RAM_EXEC 0x10000
    {
        * (+RW, +ZI)
    }

    HEAP +0 EMPTY 0x3000
    {
    }

    STACKS 0x20000 EMPTY -0x3000
    {
    }

    CODE_ONE 0x08400000 OVERLAY 0x4000
    {
        overlay_one.o (+RO)
    }
```

```
    CODE_TWO 0x08400000 OVERLAY 0x4000
    {
        overlay_two.o (+RO)
    }

    DATA_ONE 0x08700000 OVERLAY 0x4000
    {
        overlay_one.o (+RW,+ZI)
    }

    DATA_TWO 0x08700000 OVERLAY 0x4000
    {
        overlay_two.o (+RW,+ZI)
    }

}
```

8. Build the example application:

```
armclang -c -g -target arm-arm-none-eabi -mcpu=cortex-a9 -O0 main.c overlay_stubs.c
overlay_manager.c retarget.c
armclang -c -g -target arm-arm-none-eabi -mcpu=cortex-a9 -O0 func1.c -o overlay_one.o
armclang -c -g -target arm-arm-none-eabi -mcpu=cortex-a9 -O0 func2.c -o overlay_two.o
armasm --debug --cpu=cortex-a9 --keep overlay_list.s
armlink --cpu=cortex-a9 --datacompressor=off --scatter embedded_scat.scat main.o
overlay_one.o overlay_two.o overlay_stubs.o overlay_manager.o overlay_list.o retarget.o -
o image.axf
```

Related concepts

8.3 Manual overlay support on page 8-73.

Related information

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

Related concepts

8.1 Overlay support in ARM® Compiler on page 8-67.

Related information

Execution region attributes.

--emit_debug_overlay_relocs linker option.

--emit_debug_overlay_section linker option.

Chapter 9
Building Secure and Non-secure Images Using ARMv8-M Security Extensions

Describes how to use the ARMv8-M Security Extensions to build a secure image, and how to allow a non-secure image to call a secure image.

It contains the following sections:

9.1 Overview of building Secure and Non-secure images

ARM Compiler 6 tools allow you to build images that run in the Secure state of the ARMv8-M Security Extensions. You can also create an import library package that developers of Non-secure images must have for those images to call the Secure image.

Note

ARMv8-M Security Extensions are not supported when building *Read-Only Position-Independent* (ROPI) and *Read-Write Position-Independent* (RWPI) images.

To build an image that runs in the Secure state you must include the `<arm_cmse.h>` header in your code, and compile using the `armclang -mcmse` command-line option. Compiling in this way makes the following features available:

- The Test Target, `TT`, instruction.
- `TT` instruction intrinsics.
- Non-secure function pointer intrinsics.
- The `__attribute__((cmse_nonsecure_call))` and `__attribute__((cmse_nonsecure_entry))` function attributes.

On startup, your Secure code must set up the *Security Attribution Unit* (SAU) and call the Non-secure startup code.

Important considerations when compiling Secure and Non-secure code

Be aware of the following when compiling Secure and Non-secure code:

- You can compile your Secure and Non-secure code in C or C++, but the boundary between the two must have C function call linkage.
- You cannot pass C++ objects, such as classes and references, across the security boundary.
- You must not throw C++ exceptions across the security boundary.
- The value of the `__ARM_FEATURE_CMSE` predefined macro indicates what ARMv8-M Security Extension features are supported.
- Compile Secure code with the maximum capabilities for the target. For example, if you compile with no FPU then the Secure functions do not clear floating-point registers when returning from functions declared as `__attribute__((cmse_nonsecure_entry))`. Therefore, the functions could potentially leak sensitive data.
- Structs with undefined bits caused by padding and half float are currently unsupported as arguments and return values for Secure functions. Using such structs might leak sensitive information. Structs that are large enough to be passed by pointer are also unsupported and produce an error.
- The following cases are not supported when compiling with `-mcmse` and give an error:
  - Variadic entry functions.
  - Entry functions with arguments that do not fit in registers, because there are either many arguments or the arguments have large values.
  - Non-secure function calls with arguments that do not fit in registers, because there are either many arguments or the arguments have large values.

How a Non-secure image calls a Secure image using veneers

Calling a Secure image from a Non-secure image requires a transition from Non-secure to Secure state. A transition is initiated through Secure gateway veneers. Secure gateway veneers decouple the addresses from the rest of the Secure code.

An entry point in the Secure image, *`entryname`*, is identified with:

```
__acle_se_entryname:
entryname:
```

The calling sequence is as follows:

1. The Non-secure image uses the branch `BL` instruction to call the Secure gateway veneer for the required entry function in the Secure image:

```
bl    entryname
```

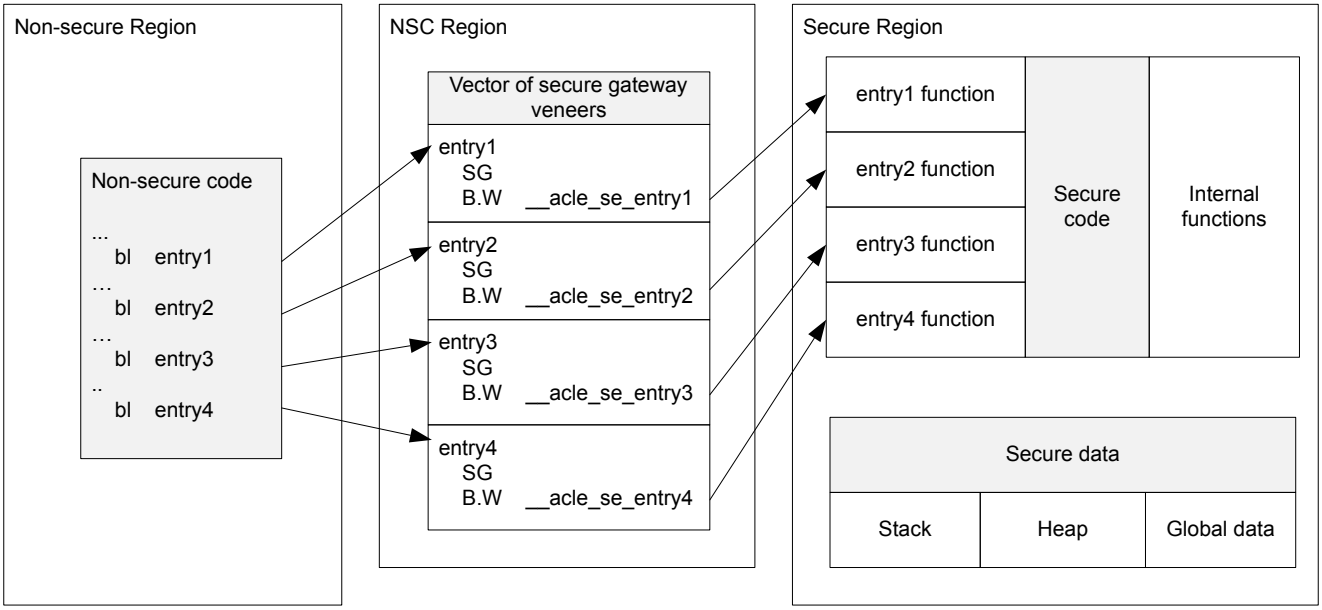
2. The Secure gateway veneer consists of the `SG` instruction and a call to the entry function in the Secure image using the `B` instruction:

```
entryname    SG
    B.W       __acle_se_entryname
```

3. The Secure image returns from the entry function using the `BXNS` instruction:

```
bxns  lr
```

The following figure is a graphical representation of the calling sequence, but for clarity, the return from the entry function is not shown:



Import library package

An import library package identifies the entry functions available in a Secure image. The import library package contains:

- An interface header file, for example `myinterface.h`. You manually create this file using any text editor.
- An import library, for example `importlib.o`. `armlink` generates this library during the link stage for a Secure image.

  Note

  You must do separate compile and link stages:

  — To create an import library when building a Secure image.
  — To use an import library when building a Non-secure image.

Related tasks

9.2 Building a Secure image using the ARMv8-M Security Extensions on page 9-84.
9.4 Building a Secure image using a previously generated import library on page 9-90.
9.3 Building a Non-secure image that can call a Secure image on page 9-88.

Related information

Whitepaper - ARMv8-M Architecture Technical Overview.
-mcmse.
__attribute__((cmse_nonsecure_call)) function attribute.
__attribute__((cmse_nonsecure_entry)) function attribute.
Predefined macros.
TT instruction intrinsics.
Non-secure function pointer intrinsics.
B instruction.
BL instruction.
BXNS instruction.
SG instruction.
TT, TTT, TTA, TTAT instruction.
Placement of CMSE veneer sections for a Secure image.

9.2 Building a Secure image using the ARMv8-M Security Extensions

When building a Secure image you must also generate an import library that specifies the entry points to the Secure image. The import library is used when building a Non-secure image that needs to call the Secure image.

Prerequisites

The following procedure is not a complete example, and assumes that your code sets up the *Security Attribution Unit* (SAU) and calls the Non-secure startup code.

Procedure

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

```
#ifdef __cplusplus
extern "C" {
#endif

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

#ifdef __cplusplus
}
#endif
```

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

```
#include <arm_cmse.h>
#include "myinterface_v1.h"

int func1(int x) { return x; }
int __attribute__((cmse_nonsecure_entry)) entry1(int x) { return func1(x);  }
int __attribute__((cmse_nonsecure_entry)) entry2(int x) { return entry1(x); }

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

   In addition to the implementation of the two entry functions, the code defines the function `func1()` that is called only by Secure code.

   Note

   If you are compiling the Secure code as C++, then you must add `extern "C"` to the functions declared as `__attribute__((cmse_nonsecure_entry))`.

3. Create an object file using the `armclang -mcmse` command-line options:

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

4. Enter the following command to see the disassembly of the machine code that `armclang` generates:

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

   The disassembly is stored in the file `secure.s`, for example:

```
    .text
    ...
    .code 16
    .thumb_func
    ...
func1:
    .fnstart
    ...
    bx lr
    ...
__acle_se_entry1:
entry1:
    .fnstart
@ BB#0:
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl func1
```

```
    ...
    pop.w {r7, lr}
    ...
    bxns lr
    ...
__acle_se_entry2:
entry2:
    .fnstart
@ BB#0:
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl entry1
    ...
    pop.w {r7, lr}
    bxns lr
    ...
main:
    .fnstart
@ BB#0:
    ...
    movs r0, #0
    ...
    bx lr
    ...
```

An entry function does not start with a Secure Gateway (`SG`) instruction. The two symbols `__acle_se_`*`entry_name`* and *`entry_name`* indicate the start of an entry function to the linker.

5. Create a scatter file containing the `Veneer$$CMSE` selector to place the entry function veneers in a *Non-Secure Callable* (NSC) memory region.

```
LOAD_REGION 0x0 0x3000
{
    EXEC_R 0x0
    {
        *(+RO,+RW,+ZI)
    }
    EXEC_NSCR 0x4000 0x1000
    {
        *(Veneer$$CMSE)
    }
    ARM_LIB_STACK 0x700000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
...
```

6. Link the object file using the `armlink --import-cmse-lib-out` command-line option and the scatter file to create the Secure image:

```
$ armlink secure.o -o secure.axf --cpu 8-M.Main --import-cmse-lib-out importlib_v1.o --
scatter secure.scf
```

In addition to the final image, the link in this example also produces the import library, `importlib_v1.o`, for use when building a Non-secure image. Assuming that the section with veneers is placed at address `0x4000`, the import library consists of a relocatable file containing only a symbol table with the following entries:

Symbol type	Name	Address
STB_GLOBAL, SHN_ABS, STT_FUNC	entry1	0x4001
STB_GLOBAL, SHN_ABS, STT_FUNC	entry2	0x4009

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

——— Note ———

If you have an import library from a previous build of the Secure image, you can ensure that the addresses in the output import library do not change when producing a new version of the Secure image. To ensure that the addresses do not change, specify the `--import-cmse-lib-in` command-

line option together with the `--import-cmse-lib-out` option. However, make sure the input and output libraries have different names.

7. Enter the following command to see the entry veneers that the linker generates:

```
$ fromelf --text -s -c secure.axf
```

The following entry veneers are generated in the EXEC_NSCR *execute-only* (XO) region for this example:

```
...

** Section #3 'EXEC_NSCR' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
        0x00004000:    e97fe97f    ....    SG        ; [0x3e08]
        0x00004004:    f7fcb85e    ..^.    B         __acle_se_entry1 ; 0xc4
    entry2
        0x00004008:    e97fe97f    ....    SG        ; [0x3e10]
        0x0000400c:    f7fcb86c    ..l.    B         __acle_se_entry2 ; 0xe8

...
```

The section with the veneers is aligned on a 32-byte boundary and padded to a 32-byte boundary.

If you do not use a scatter file, the entry veneers are placed in an ER_XO section as the first execution region, for example:

```
...

** Section #1 'ER_XO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00008000

    $t
    entry1
        0x00008000:    e97fe97f    ....    SG        ; [0x7e08]
        0x00008004:    f000b85a    ..Z.    B.W       __acle_se_entry1 ; 0x80bc
    entry2
        0x00008008:    e97fe97f    ....    SG        ; [0x7e10]
        0x0000800c:    f000b868    ..h.    B.W       __acle_se_entry2 ; 0x80e0

...
```

Postrequisites

After you have built your Secure image:

1. Pre-load the Secure image onto your device.
2. Deliver your device with the pre-loaded image, together with the import library package, to a party who develops the Non-secure code for this device. The import library package contains:
   - The interface header file, `myinterface_v1.h`.
   - The import library, `importlib_v1.o`.

Related tasks

9.4 Building a Secure image using a previously generated import library on page 9-90.

9.3 Building a Non-secure image that can call a Secure image on page 9-88.

Related information

Whitepaper - ARMv8-M Architecture Technical Overview.

-c armclang option.

-march armclang option.

-mcmse armclang option.

-S armclang option.

--target armclang option.

__attribute__((cmse_nonsecure_entry)) function attribute.
SG instruction.
--cpu armlink option.
--import_cmse_lib_in armlink option.
--import_cmse_lib_out armlink option.
--scatter armlink option.
--text fromelf option.

9.3 Building a Non-secure image that can call a Secure image

If you are building a Non-secure image that is to call a Secure image, the Non-secure code must be written in C. You must also obtain the import library package that was created for that Secure image.

Prerequisites

The following procedure assumes that you have the import library package that is created in *9.2 Building a Secure image using the ARMv8-M Security Extensions* on page 9-84. The package provides the C linkage that allows you to compile your Non-secure code as C or C++.

The import library package identifies the entry points for the Secure image.

Procedure

1. Include the interface header file in the C program for your Non-secure code, `nonsecure.c`, and use the entry functions as required, for example:

```
#include <stdio.h>
#include "myinterface_v1.h"

int main(void) {
    int val1, val2, x;

    val1 = entry1(x);
    val2 = entry2(x);

    if (val1 == val2) {
        printf("val2 is equal to val1\n");
    } else {
        printf("val2 is different from val1\n");
    }

    return 0;
}
```

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

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

3. Create a scatter file for the Non-secure image, but without the *Non-Secure Callable* (NSC) memory region, for example:

```
LOAD_REGION 0x8000 0x3000
{
    ER 0x8000
    {
        *(+RO,+RW,+ZI)
    }
    ARM_LIB_STACK 0x800000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
...
```

4. Link the object file using the import library, `importlib_v1.o`, and the scatter file to create the Non-secure image:

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

Related tasks

9.2 Building a Secure image using the ARMv8-M Security Extensions on page 9-84.

Related information

Whitepaper - ARMv8-M Architecture Technical Overview.

-march armclang option.

--target armclang option.
--cpu armlink option.
--scatter armlink option.

9.4 Building a Secure image using a previously generated import library

You can build a new version of a Secure image and use the same addresses for the entry points that were present in the previous version. You specify the import library that is generated for the previous version of the Secure image and generate another import library for the new Secure image.

Prerequisites

The following procedure is not a complete example, and assumes that your code sets up the *Security Attribution Unit* (SAU) and calls the Non-secure startup code.

The following procedure assumes that you have the import library package that is created in *9.2 Building a Secure image using the ARMv8-M Security Extensions* on page 9-84.

Procedure

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

```
#ifdef __cplusplus
extern "C" {
#endif

int entry1(int x);
int entry2(int x);
int entry3(int x);
int entry4(int x);

#ifdef __cplusplus
}
#endif
```

2. Include the following in the C program for your Secure code, `secure.c`:

```
#include <arm_cmse.h>
#include "myinterface_v2.h"

int func1(int x) { return x; }
int __attribute__((cmse_nonsecure_entry)) entry1(int x) { return func1(x);  }
int __attribute__((cmse_nonsecure_entry)) entry2(int x) { return entry1(x); }
int __attribute__((cmse_nonsecure_entry)) entry3(int x) { return func1(x) + entry1(x);  }
int __attribute__((cmse_nonsecure_entry)) entry4(int x) { return entry1(x) * entry2(x); }

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

   In addition to the implementation of the two entry functions, the code defines the function `func1()` that is called only by Secure code.

   Note

   If you are compiling the Secure code as C++, then you must add `extern "C"` to the functions declared as `__attribute__((cmse_nonsecure_entry))`.

3. Create an object file using the `armclang -mcmse` command-line options:

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

4. To see the disassembly of the machine code that is generated by `armclang`, enter:

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

   The disassembly is stored in the file `secure.s`, for example:

```
    .text
    ...
    .code 16
    .thumb_func

    ...

func1:
    .fnstart
    ...
```

```
    bx lr

    ...

__acle_se_entry1:
entry1:
    .fnstart
@ BB#0:
    .save    {r7, lr}
    push    {r7, lr}
    ...
    bl func1
    pop.w {r7, lr}
    ...
    bxns lr

    ...

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

    ...

main:
    .fnstart
@ BB#0:
    ...
    movs r0, #0
    ...
    bx lr

    ...
```

An entry function does not start with a Secure Gateway (`SG`) instruction. The two symbols `__acle_se_`*`entry_name`* and *`entry_name`* indicate the start of an entry function to the linker.

5. Create a scatter file containing the `Veneer$$CMSE` selector to place the entry function veneers in a *Non-Secure Callable* (NSC) memory region.

```
LOAD_REGION 0x0 0x3000
{
    EXEC_R 0x0
    {
        *(+RO,+RW,+ZI)
    }
    EXEC_NSCR 0x4000 0x1000
    {
        *(Veneer$$CMSE)
    }
    ARM_LIB_STACK 0x700000 EMPTY -0x10000
    {
    }
    ARM_LIB_HEAP  +0 EMPTY 0x10000
    {
    }
}
...
```

6. Link the object file using the `armlink --import-cmse-lib-out` and `--import-cmse-lib-in` command-line option, together with the preprocessed scatter file to create the Secure image:

```
$ armlink secure.o -o secure.axf --cpu 8-M.Main --import-cmse-lib-out importlib_v2.o --
import-cmse-lib-in importlib_v1.o --scatter secure.scf
```

In addition to the final image, the link in this example also produces the import library, `importlib_v2.o`, for use when building a Non-secure image. Assuming that the section with veneers is placed at address `0x4000`, the import library consists of a relocatable file containing only a symbol table with the following entries:

Symbol type	Name	Address
STB_GLOBAL, SHN_ABS, STT_FUNC	entry1	0x4001
STB_GLOBAL, SHN_ABS, STT_FUNC	entry2	0x4009
STB_GLOBAL, SHN_ABS, STT_FUNC	entry3	0x4021
STB_GLOBAL, SHN_ABS, STT_FUNC	entry4	0x4029

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

7. Enter the following command to see the entry veneers that the linker generates:

```
$ fromelf --text -s -c secure.axf
```

The following entry veneers are generated in the EXEC_NSCR *execute-only* (XO) region for this example:

```
...

** Section #3 'EXEC_NSCR' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 64 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
        0x00004000:    e97fe97f    ....    SG        ; [0x3e08]
        0x00004004:    f7fcb85e    ..^.    B         __acle_se_entry1 ; 0xc4
    entry2
        0x00004008:    e97fe97f    ....    SG        ; [0x3e10]
        0x0000400c:    f7fcb86c    ..l.    B         __acle_se_entry2 ; 0xe8

...

    entry3
        0x00004020:    e97fe97f    ....    SG        ; [0x3e28]
        0x00004024:    f7fcb872    ..r.    B         __acle_se_entry3 ; 0x10c
    entry4
        0x00004028:    e97fe97f    ....    SG        ; [0x3e30]
        0x0000402c:    f7fcb888    ....    B         __acle_se_entry4 ; 0x140

...
```

The section with the veneers is aligned on a 32-byte boundary and padded to a 32-byte boundary.

If you do not use a scatter file, the entry veneers are placed in an ER_XO section as the first execution region. The entry veneers for the existing entry points are placed in a CMSE veneer section. For example:

```
...

** Section #1 'ER_XO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00008000

    $t
    entry3
        0x00008000:    e97fe97f    ....    SG        ; [0x7e08]
        0x00008004:    f000b87e    ..~.    B.W       __acle_se_entry3 ; 0x8104
    entry4
        0x00008008:    e97fe97f    ....    SG        ; [0x7e10]
        0x0000800c:    f000b894    ....    B.W       __acle_se_entry4 ; 0x8138

...

** Section #4 'ER$$Veneer$$CMSE_AT_0x00004000' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR
+ SHF_ARM_NOREAD]
    Size   : 32 bytes (alignment 32)
    Address: 0x00004000

    $t
    entry1
        0x00004000:    e97fe97f    ....    SG        ; [0x3e08]
        0x00004004:    f004b85a    ..Z.    B.W       __acle_se_entry1 ; 0x80bc
    entry2
        0x00004008:    e97fe97f    ....    SG        ; [0x3e10]
        0x0000400c:    f004b868    ..h.    B.W       __acle_se_entry2 ; 0x80e0
```

```
...
```

Postrequisites

After you have built your updated Secure image:

1. Pre-load the updated Secure image onto your device.
2. Deliver your device with the pre-loaded image, together with the new import library package, to a party who develops the Non-secure code for this device. The import library package contains:
   - The interface header file, `myinterface_v2.h`.
   - The import library, `importlib_v2.o`.

Related tasks

9.2 Building a Secure image using the ARMv8-M Security Extensions on page 9-84.

9.3 Building a Non-secure image that can call a Secure image on page 9-88.

Related information

Whitepaper - ARMv8-M Architecture Technical Overview.

-c armclang option.

-march armclang option.

-mcmse armclang option.

-S armclang option.

--target armclang option.

__attribute__((cmse_nonsecure_entry)) function attribute.

SG instruction.

--cpu armlink option.

--import_cmse_lib_in armlink option.

--import_cmse_lib_out armlink option.

--scatter armlink option.

--text fromelf option.