Arm® Compiler

Version 6.16

Migration and Compatibility Guide

100068_0616_01_en

Arm® Compiler

Migration and Compatibility 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
0607-00	05 April 2017	Non-Confidential	Arm Compiler v6.7 Release. Document numbering scheme has changed.
0608-00	30 July 2017	Non-Confidential	Arm Compiler v6.8 Release.
0609-00	25 October 2017	Non-Confidential	Arm Compiler v6.9 Release.
0610-00	14 March 2018	Non-Confidential	Arm Compiler v6.10 Release.
0611-00	25 October 2018	Non-Confidential	Arm Compiler v6.11 Release.
0612-00	27 February 2019	Non-Confidential	Arm Compiler v6.12 Release.
0613-00	09 October 2019	Non-Confidential	Arm Compiler v6.13 Release.
0614-00	26 February 2020	Non-Confidential	Arm Compiler v6.14 Release.
0615-00	07 October 2020	Non-Confidential	Arm Compiler v6.15 Release.
0615-01	14 December 2020	Non-Confidential	Documentation update 1 for Arm Compiler v6.15 Release.
0616-00	03 March 2021	Non-Confidential	Arm Compiler v6.16 Release.
0616-01	12 March 2021	Non-Confidential	Documentation update 1 for Arm Compiler v6.16 Release.

Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING

OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

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

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

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at *http://www.arm.com/company/policies/trademarks*.

Copyright © 2014–2021 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

developer.arm.com

Progressive terminology commitment

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

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

Contents

Arm® Compiler Migration and Compatibility Guide

Preface

Chapter 1 Configuration and Support Information

Chapter 2 Migrating from Arm® Compiler 5 to Arm® Compiler 6

Chapter 3 Migrating from armcc to armclang

List of Figures
Arm® Compiler Migration and Compatibility Guide

List of Tables
Arm® Compiler Migration and Compatibility Guide

Preface

This preface introduces the *Arm® Compiler Migration and Compatibility Guide*.

It contains the following:

- *About this book* on page 11.

About this book

The Arm® Compiler Migration and Compatibility Guide provides migration and compatibility information for users moving from older versions of Arm Compiler to Arm Compiler 6.

Using this book

This book is organized into the following chapters:

Chapter 1 Configuration and Support Information
A summary of the support levels and FlexNet versions supported by the Arm compilation tools.

Chapter 2 Migrating from Arm® Compiler 5 to Arm® Compiler 6
Provides an overview of the differences between Arm Compiler 5 and Arm Compiler 6.

Chapter 3 Migrating from armcc to armclang
Compares Arm Compiler 6 command-line options to older versions of Arm Compiler.

Chapter 4 Compiler Source Code Compatibility
Provides details of source code compatibility between Arm Compiler 6 and older `armcc` compiler versions.

Chapter 5 Migrating from armasm to the armclang Integrated Assembler
Describes how to migrate assembly code from `armasm` syntax to GNU syntax (used by `armclang`).

Chapter 6 Changes Between Different Versions of Arm® Compiler 6
A description of the changes that affect migration and compatibility between different versions of Arm Compiler 6.

Appendix A Code Examples
Provides source code examples for Arm Compiler 5 and Arm Compiler 6.

Appendix B Licenses
Describes the Apache license.

Appendix C Arm® Compiler Migration and Compatibility Guide Changes
Describes the technical changes that have been made to the Arm Compiler Migration and Compatibility Guide.

Glossary

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

See the *Arm® Glossary* for more information.

Typographic conventions

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

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

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

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

`monospace italic`

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

`monospace bold`

Denotes language keywords when used outside example code.

`<and>`

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

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

SMALL CAPITALS

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

Feedback

Feedback on this product

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

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

Feedback on content

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

- The title *Arm Compiler Migration and Compatibility Guide*.
- The number 100068_0616_01_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

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

Other information

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

Chapter 1

Configuration and Support Information

A summary of the support levels and FlexNet versions supported by the Arm compilation tools.

It contains the following sections:

1.1 Support level definitions

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

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

Arm welcomes feedback regarding the use of all Arm Compiler 6 features, and intends to support users to a level that is appropriate for that feature. You can contact support at *https://developer.arm.com/support*.

Identification in the documentation

All features that are documented in the Arm Compiler 6 documentation are product features, except where explicitly stated. The limitations of non-product features are explicitly stated.

Product features

Product features are suitable for use in a production environment. The functionality is well-tested, and is expected to be stable across feature and update releases.

- Arm intends to give advance notice of significant functionality changes to product features.
- If you have a support and maintenance contract, Arm provides full support for use of all product features.
- Arm welcomes feedback on product features.
- Any issues with product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

In addition to fully supported product features, some product features are only alpha or beta quality.

Beta product features

Beta product features are implementation complete, but have not been sufficiently tested to be regarded as suitable for use in production environments.

Beta product features are indicated with [BETA].

- Arm endeavors to document known limitations on beta product features.
- Beta product features are expected to eventually become product features in a future release of Arm Compiler 6.
- Arm encourages the use of beta product features, and welcomes feedback on them.
- Any issues with beta product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

Alpha product features

Alpha product features are not implementation complete, and are subject to change in future releases, therefore the stability level is lower than in beta product features.

Alpha product features are indicated with [ALPHA].

- Arm endeavors to document known limitations of alpha product features.
- Arm encourages the use of alpha product features, and welcomes feedback on them.
- Any issues with alpha product features that Arm encounters or is made aware of are considered for fixing in future versions of Arm Compiler.

Community features

Arm Compiler 6 is built on LLVM technology and preserves the functionality of that technology where possible. This means that there are additional features available in Arm Compiler that are not listed in the documentation. These additional features are known as community features. For information on these community features, see the *documentation for the Clang/LLVM project*.

Where community features are referenced in the documentation, they are indicated with [COMMUNITY].

- Arm makes no claims about the quality level or the degree of functionality of these features, except when explicitly stated in this documentation.
- Functionality might change significantly between feature releases.
- Arm makes no guarantees that community features will remain functional across update releases, although changes are expected to be unlikely.

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

Guidance on use of community features

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

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

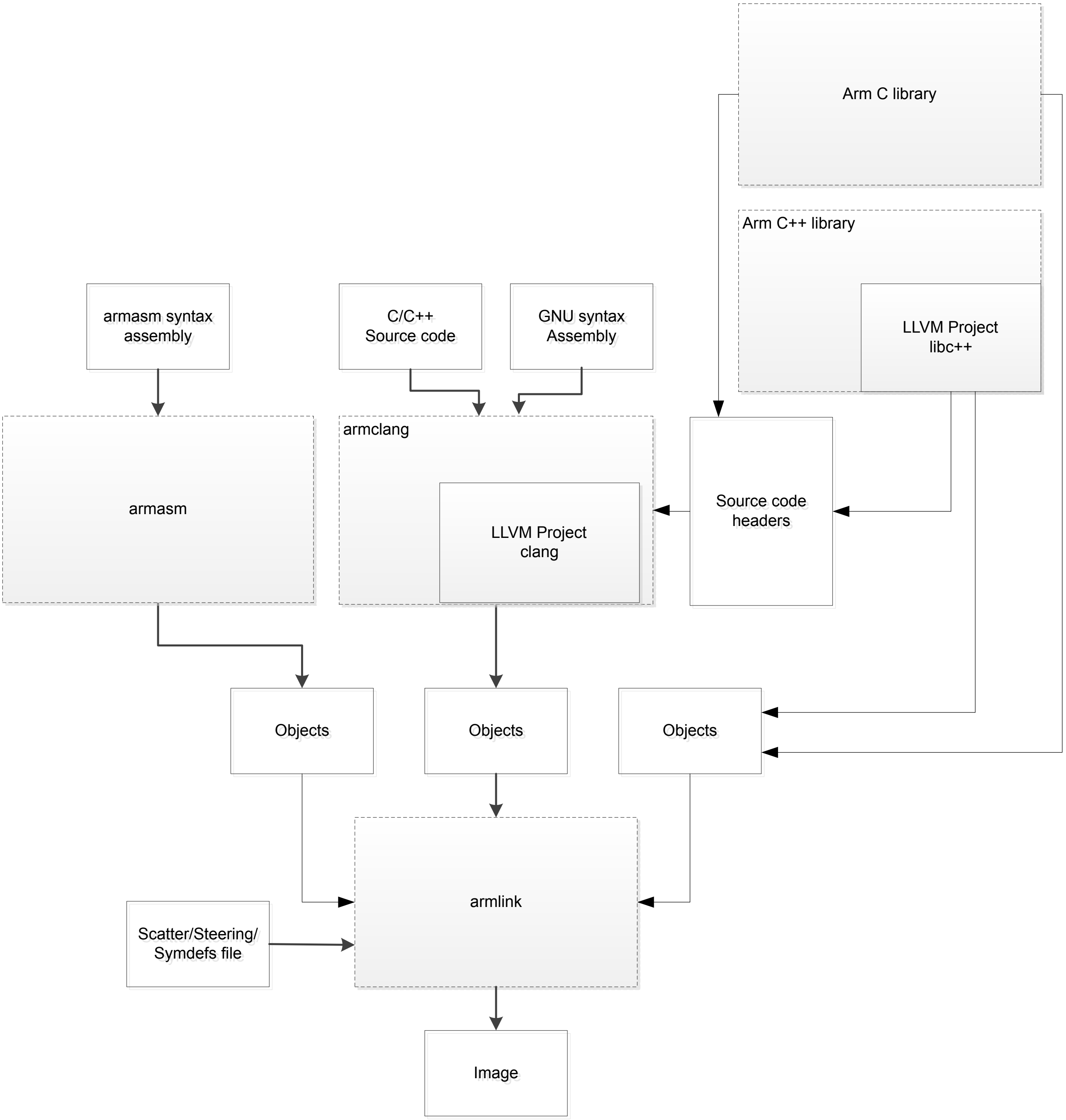

Figure 1-1 Integration boundaries in Arm Compiler 6.

The dashed boxes are toolchain components, and any interaction between these components is an integration boundary. Community features that span an integration boundary might have significant limitations in functionality. The exception to this is if the interaction is codified in one of the standards supported by Arm Compiler 6. See *Application Binary Interface (ABI) for the Arm® Architecture*. Community features that do not span integration boundaries are more likely to work as expected.

- Features primarily used when targeting hosted environments such as Linux or BSD might have significant limitations, or might not be applicable, when targeting bare-metal environments.
- The Clang implementations of compiler features, particularly those that have been present for a long time in other toolchains, are likely to be mature. The functionality of new features, such as support

for new language features, is likely to be less mature and therefore more likely to have limited functionality.

Deprecated features

A deprecated feature is one that Arm plans to remove from a future release of Arm Compiler. Arm does not make any guarantee regarding the testing or maintenance of deprecated features. Therefore, Arm does not recommend using a feature after it is deprecated.

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

Unsupported features

With both the product and community feature categories, specific features and use-cases are known not to function correctly, or are not intended for use with Arm Compiler 6.

Limitations of product features are stated in the documentation. Arm cannot provide an exhaustive list of unsupported features or use-cases for community features. The known limitations on community features are listed in *Community features* on page 1-14.

List of known unsupported features

The following is an incomplete list of unsupported features, and might change over time:

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

  Note

  This restriction does not apply to the [ALPHA]-supported multithreaded C++ libraries.

- Use of C11 library features is unsupported.
- Any community feature that is exclusively related to non-Arm architectures is not supported.
- Compilation for targets that implement architectures older than Armv7 or Armv6-M is not supported.
- The **`long double`** data type is not supported for AArch64 state because of limitations in the current Arm C library.
- C complex arithmetic is not supported, because of limitations in the current Arm C library.
- C++ complex numbers are supported with **`float`** and **`double`** types, but not **`long double`** type because of limitations in the current Arm C library.

1.2 Compiler configuration information

Summarizes the FlexNet versions supported by the Arm compilation tools.

FlexNet versions in the compilation tools

Different versions of Arm Compiler support different versions of FlexNet.

The FlexNet versions in the compilation tools are:

Table 1-1 FlexNet versions

Compilation tools version	Windows	Linux
Arm Compiler 6.7 and later	11.14.1.0	11.14.1.0
Arm Compiler 6.01 and later	11.12.1.0	11.12.1.0
Arm Compiler 6.00	11.10.1.0	11.10.1.0

Related information

Arm DS-5 License Management Guide

Chapter 2

Migrating from Arm® Compiler 5 to Arm® Compiler 6

Provides an overview of the differences between Arm Compiler 5 and Arm Compiler 6.

It contains the following sections:

2.1 Migration overview

Migrating from Arm Compiler 5 to Arm Compiler 6 requires the use of new command-line options and might also require changes to existing source files.

Arm Compiler 6 is based on the modern LLVM compiler framework. Arm Compiler 5 is not based on the LLVM compiler framework. Therefore migrating your project and source files from Arm Compiler 5 to Arm Compiler 6 requires you to be aware of:

- Differences in the command-line options when invoking the compiler.
- Differences in the adherence to language standards.
- Differences in compiler specific keywords, attributes, and pragmas.
- Differences in optimization and diagnostic behavior of the compiler.

Even though these differences exist between Arm Compiler 5 and Arm Compiler 6, it is possible to migrate your projects from Arm Compiler 5 to Arm Compiler 6 by modifying your command-line arguments and by changing your source code if required.

Arm Compiler 5 does not support processors based on Armv8 and later architectures. Migrating to Arm Compiler 6 enables you to generate highly efficient code for processors based on Armv8 and later architectures.

Related references

2.4 Optimization differences on page 2-24

2.6 Diagnostic messages on page 2-27

3.1 Migration of compiler command-line options from Arm® Compiler 5 to Arm® Compiler 6 on page 3-32

Chapter 4 Compiler Source Code Compatibility on page 4-54

Related information

Migrating projects from Arm Compiler 5 to Arm Compiler 6

2.2 Toolchain differences

Arm Compiler 5 and Arm Compiler 6 share many of the same compilation tools. However, the main difference between the two toolchains is the compiler tool `armclang`, which is based on Clang and LLVM.

The table lists the individual compilation tools and the toolchain they apply to.

Table 2-1 List of compilation tools

Arm Compiler 5	Arm Compiler 6	Function
`armcc`	`armclang`	Compiles C and C++ language source files, including inline assembly.
`armcc`	`armclang`	Preprocessor.
`armasm`	`armasm`	Legacy assembler for assembly language source files written in `armasm` syntax. Use the `armclang` integrated assembler for all new assembly files.
Not available	`armclang`. This is also called the `armclang` integrated assembler.	Assembles assembly language source files written in GNU assembly syntax.
`fromelf`	`fromelf`	Converts Arm ELF images to binary formats and can also generate textual information about the input image, such as its disassembly and its code and data size.
`armlink`	`armlink`	Combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program.
`armar`	`armar`	Enables sets of ELF object files to be collected together and maintained in archives or libraries.

Arm Compiler 6 uses the compiler tool `armclang` instead of `armcc`. The command-line options for `armclang` are different to the command-line options for `armcc`. These differences are described in *3.1 Migration of compiler command-line options from Arm® Compiler 5 to Arm® Compiler 6* on page 3-32.

Arm Compiler 6 includes the legacy assembler `armasm`, which you can use to assemble your older assembly language source files if they are written in `armasm` syntax. Arm recommends that you write new assembly code using the GNU assembly syntax, which you can assemble using the `armclang` integrated assembler. You can also migrate existing assembly language source files from `armasm` syntax to GNU syntax, and then assemble them using the `armclang` integrated assembler. For more information see *Chapter 5 Migrating from armasm to the armclang Integrated Assembler* on page 5-75.

Related information

Migrating projects from Arm Compiler 5 to Arm Compiler 6

2.3 Default differences

Some compiler and assembler options are different between Arm Compiler 5 and Arm Compiler 6, or have different default values.

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page 1-14.

The following table lists these differences.

Table 2-2 Differences in defaults

Arm Compiler 5	Arm Compiler 6	Notes	Further information
`--hide_all`	`-fvisibility=hidden`	These defaults are similar but `-fvisibility=hidden` does not affect extern declarations or symbol references. In Arm Compiler 6, symbols in the final image are hidden if the reference or the definition is hidden. Therefore the visibility of the reference alone does not determine the visibility of the symbol, as it does in Arm Compiler 5.	*`--hide_all` for Arm® Compiler 5.* *`-fvisibility` for Arm® Compiler 6.*
`--apcs=/hardfp` or `--apcs=/softfp`	`-mfloat-abi=softfp`	The default floating-point linkage in Arm Compiler 5 depends on the specified processor. If the processor has floating-point hardware, then Arm Compiler 5 uses hardware floating-point linkage. If the processor does not have floating-point hardware, then Arm Compiler 5 uses software floating-point linkage. In Arm Compiler 6, the default is always software floating-point linkage for AArch32 state. The `-mfloat-abi` option also controls the type of floating-point instructions that the compiler uses. `-mfloat-abi=softfp` uses hardware floating-point instructions. Use `-mfloat-abi=soft` to use software floating-point linkage and software library functions for floating-point operations.	*`--apcs` for Arm® Compiler 5.* *`-mfloat-abi` for Arm® Compiler 6.*
`__image.axf`	`a.out`	Default name for the executable image if none of `-o`, `-c`, `-E`, or `-S` are specified on the command-line.	*`-o` for Arm® Compiler 5.* *`-o` for Arm® Compiler 6.*
`--enum_is_int` is disabled by default	`-fno-short-enums`	`--enum_is_int` is disabled by default in Arm Compiler 5, so the smallest data type that can hold the enumerator values is used. `-fno-short-enums` is the default in Arm Compiler 6, so the size of the enumeration type is at least 32 bits.	*`--enum_is_int` for Arm® Compiler 5.* *`-fno-short-enums` for Arm® Compiler 6.*

Table 2-2 Differences in defaults (continued)

Arm Compiler 5	Arm Compiler 6	Notes	Further information
`-O2`	`-O0`	Arm Compiler 5 uses high optimization (`-O2`) by default. Arm Compiler 6 uses minimum optimization (`-O0`) by default.	*`-O` for Arm® Compiler 5.* *`-O` for Arm® Compiler 6.* *Optimization differences* on page 2-24.
`--apcs=/nointerwork`	`--apcs=/interwork`	In Arm Compiler 5, `armasm` does not specify by default that code in the input file can safely interwork between A32 and T32. In Arm Compiler 6, `armasm` specifies interworking by default for AArch32 targets that support A32 and T32 instruction sets.	*`--apcs` for Arm® Compiler 5.* *`--apcs` for Arm® Compiler 6.*
C++03	C++14	In Arm Compiler 5, the default C++ source language mode is C++03. In Arm Compiler 6, the default C++ source language mode is C++14. You can override the default source language with `-std` in Arm Compiler 6.	*`--cpp` for Arm® Compiler 5.* *`-std` for Arm® Compiler 6.*
C90	C11 [COMMUNITY]	In Arm Compiler 5, the default C source language mode is C90. In Arm Compiler 6, the default C source language mode is C11 [COMMUNITY]. You can override the default source language with `-std` in Arm Compiler 6.	*`--c90` for Arm® Compiler 5.* *`-std` for Arm® Compiler 6.*
`--no_exceptions`	`-fexceptions` or `-fno-exceptions`	In Arm Compiler 5, C++ exceptions are disabled by default (`--no_exceptions`). In Arm Compiler 6, C++ exceptions are enabled by default (`-fexceptions`) for C++ sources, or disabled by default (`-fno-exceptions`) for C sources.	*`--no_exceptions` for Arm® Compiler 5.* *`-fexceptions` for Arm® Compiler 6.*
`--wchar16`	`-fno-short-wchar`	In Arm Compiler 5, the size of `wchar_t` is 2 bytes by default (`--wchar16`). In Arm Compiler 6, the size of `wchar_t` is 4 bytes by default (`-fno-short-wchar`).	*`--wchar16` for Arm® Compiler 5.* *`-fno-short-wchar` for Arm® Compiler 6.*
`--split_sections` is disabled by default	`-ffunction-sections`	In Arm Compiler 5, functions are not put into separate ELF sections by default (`--split_sections` is disabled). In Arm Compiler 6, each function is put into a separate ELF section by default (`-ffunction-sections`).	*`--split_sections` for Arm® Compiler 5.* *`-ffunction-sections` for Arm® Compiler 6.*

2.4 Optimization differences

Arm Compiler 6 provides more performance optimization settings than are present in Arm Compiler 5. However, the optimizations that are performed at each optimization level might differ between the two toolchains.

The table compares the optimization settings and functions in Arm Compiler 5 and Arm Compiler 6.

Table 2-3 Optimization settings

Description	Arm Compiler 5	Arm Compiler 6	Notes
Optimization levels for performance.	• `-Otime -O0` • `-Otime -O1` • `-Otime -O2` • `-Otime -O3`	• `-O0` • `-O1` • `-O2` • `-O3` • `-Ofast` • `-Omax`	The Arm Compiler 5 `-O0` option is more similar to the Arm Compiler 6 `-O1` option than the Arm Compiler 6 `-O0` option. The Arm Compiler 6 `-Omax` option refers to maximum performance, with Link-Time Optimization (LTO) enabled.
Optimization levels for code size.	• `-Ospace -O0` • `-Ospace -O1` • `-Ospace -O2` • `-Ospace -O3`	• `-Os` • `-Oz` • `-Omin`	The Arm Compiler 5 `-O0` option is more similar to the Arm Compiler 6 `-O1` option than the Arm Compiler 6 `-O0` option. The Arm Compiler 6 `-Omin` option refers to minimum code size, with Link-Time Optimization (LTO) enabled..
Default.	`-Ospace -O2`	`-O0`	
Best trade-off between image size, performance, and debug.	`-Ospace -O2`	`-O1`	
Highest optimization for performance.	`-Otime -O3`	• `-Omax` • `-Ofast`	The `-Omax` option uses Link-Time Optimization (LTO). If LTO is not appropriate for you, use `-Ofast`.
Highest optimization for code size.	`-Ospace -O3`	• `-Omin` • `-Oz`	The `-Omin` option uses Link-Time Optimization (LTO). If LTO is not appropriate for you, use `-Oz`.

Arm Compiler 6 provides an aggressive performance optimization setting, `-Omax`, which automatically enables a feature called Link-Time Optimization. For more information, see *-flto*.

At the opposite end of the spectrum, the `-Omin` setting in Arm Compiler 6 is an aggressive code size optimization setting. This also enables Link-Time Optimization and aggressively removes unused code and data.

When using `-Omax` or `-Omin`, `armclang` can perform link-time optimizations that were not possible in Arm Compiler 5. In some cases these link-time optimizations can expose latent bugs in a program, which manifest as an image with different or unanticapted behavior. Therefore, an image built with Arm Compiler 5 might have a different behavior to the image built with Arm Compiler 6.

For example, unused variables without the `volatile` keyword might be removed when using `-Omax` or `-Omin` in Arm Compiler 6. If the unused variable is actually a volatile variable that requires the `volatile` keyword, then the removal of the variable can cause the generated image to behave unexpectedly. Since Arm Compiler 5 does not have these aggressive optimization settings, it might not have removed the unused variable, and the resulting image might behave as expected, and therefore the error in the code would be more difficult to detect.

Related information

-flto armclang option

-O armclang option

Effect of the volatile keyword on compiler optimization

Optimizing across modules with link time optimization

2.5 Backwards compatibility issues

Some Arm Compiler 5 options produce objects that are not compatible with Arm Compiler 6.

SHF_COMDEF ELF sections

Linking with legacy objects that contain ELF sections with the legacy `SHF_COMDEF` ELF section flag is deprecated. Use the `GRP_COMDAT` ELF section group instead of the legacy `SHF_COMDEF` ELF section flag by:

- Replacing the `COMDEF` section attribute of the legacy armasm syntax `AREA` directive with the `COMGROUP=`*`symbol_name`* section attribute.
- Rebuilding incompatible legacy objects using one of the following:
  - Arm Compiler 5 but with the `--dwarf3` option. Other incompatibilities might still exist.
  - Arm Compiler 6.

Related information

AREA

--dwarf3

2.6 Diagnostic messages

In general, `armclang` provides more precise and detailed diagnostic messages compared to `armcc`. Therefore you can expect to see more information about your code when using Arm Compiler 6, which can help you understand and fix your source more quickly.

`armclang` and `armcc` differ in the quality of diagnostic information they provide about your code. The following sections demonstrate some of the differences.

Assignment in condition

The following code is an example of `armclang` providing more precise information about your code. The error in this example is that the assignment operator, `=`, must be changed to the equality operator, `==`.

```
main.cpp:

#include <stdio.h>

int main()
{
  int a = 0, b = 0;
  if (a = b)
  {
    printf("Right\n");
  }
  else
  {
    printf("Wrong\n");
  }
  return 0;
}
```

Compiling this example with Arm Compiler 5 gives the message:

```
"main.cpp", line 6: Warning: #1293-D: assignment in condition
if (a = b)
    ^
```

Compiling this example with Arm Compiler 6 gives the message:

```
main.cpp:6:7: warning: using the result of an assignment as a condition without parentheses[-
Wparentheses]
 if (a = b)
      ~^~

main.cpp:6:7: note: place parentheses around the assignment to silence this warning
 if (a = b)
       ^
      (  )

main.cpp:6:7: note: use '==' to turn this assignment into an equality comparison
 if (a = b)
       ^
       ==
```

`armclang` highlights the error in the code, and also suggests two different ways to resolve the error. The warning messages highlight the specific part which requires attention from the user.

Note

When using `armclang`, it is possible to enable or disable specific warning messages. In the example above, you can enable this warning message using the `-Wparentheses` option, or disable it using the `-Wno-parentheses` option.

Automatic macro expansion

Another very useful feature of diagnostic messages in Arm Compiler 6, is the inclusion of notes about macro expansion. These notes provide useful context to help you understand diagnostic messages resulting from automatic macro expansion.

Consider the following code:

```
main.cpp:

#include <stdio.h>

#define LOG(PREFIX, MESSAGE) fprintf(stderr, "%s: %s", PREFIX, MESSAGE)
#define LOG_WARNING(MESSAGE) LOG("Warning", MESSAGE)

int main(void)
{
    LOG_WARNING(123);
}
```

The macro `LOG_WARNING` has been called with an integer argument. However, expanding the two macros, you can see that the `fprintf` function expects a string. When the macros are close together in the code it is easy to spot these errors. These errors are not easy to spot if they are defined in different part of the source code, or in other external libraries.

Compiling this example with Arm Compiler 5 `armcc main.cpp` gives the message:

```
main.cpp", line 8: Warning:  #181-D: argument is incompatible with corresponding format
string conversion
      LOG_WARNING(123);
      ^
```

Compiling this example with Arm Compiler 6 `armclang --target=arm-arm-none-eabi -march=armv8-a` gives the message:

```
main.cpp:8:14: warning: format specifies type 'char *' but the argument has type 'int' [-
Wformat]
 LOG_WARNING(123);
 ~~~~~~~~~~~~^~~

main.cpp:4:45: note: expanded from macro 'LOG_WARNING'
#define LOG_WARNING(MESSAGE) LOG("Warning", MESSAGE)
                             ~~~~~~~~~~~~~~~^~~~~~

main.cpp:3:64: note: expanded from macro 'LOG'
#define LOG(PREFIX, MESSAGE) fprintf(stderr, "%s: %s", PREFIX, MESSAGE)
                                                  ~~           ^~~~~~
```

For more information, see *4.5 Diagnostics for pragma compatibility* on page 4-68.

Note

When starting the migration from Arm Compiler 5 to Arm Compiler 6, you can expect additional diagnostic messages because `armclang` does not recognize some of the pragmas, keywords, and attributes that were specific to `armcc`. When you replace the pragmas, keywords, and attributes from Arm Compiler 5 with their Arm Compiler 6 equivalents, the majority of these diagnostic messages disappear. You might require additional code changes if there is no direct equivalent for Arm Compiler 6. For more information see *Chapter 4 Compiler Source Code Compatibility* on page 4-54.

2.7 Migration example

This topic shows you the process of migrating an example code from Arm Compiler 5 to Arm Compiler 6.

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page 1-14.

Compiling with Arm® Compiler 5

For an example startup code that builds with Arm Compiler 5, see *Example startup code for Arm® Compiler 5 project* on page Appx-A-136.

To compile this example with Arm Compiler 5, enter:

```
armcc startup_ac5.c --cpu=7-A -c
```

This command generates a compiled object file for the Armv7-A architecture.

Compiling with Arm® Compiler 6

Try to compile the `startup_ac5.c` example with Arm Compiler 6. The first step in the migration is to use the new compiler tool, `armclang`, and use the correct command-line options for `armclang`.

To compile this example with Arm Compiler 6, enter:

```
armclang --target=arm-arm-none-eabi startup_ac5.c -march=armv7-a -c -O1 -std=c90
```

The following table shows the differences in the command-line options between Arm Compiler 5 and Arm Compiler 6:

Table 2-4 Command-line changes

Description	Arm Compiler 5	Arm Compiler 6
Tool	`armcc`	`armclang`
Specifying an architecture	`--cpu=7-A`	• `-march=armv7-a` • `--target` is a mandatory option for `armclang`.
Optimization	The default optimization is `-O2`.	The default optimization is `-O0`. To get similar optimizations as the Arm Compiler 5 default, use `-O1`.
Source language mode	The default source language mode for `.c` files is `c90`.	The default source language mode for `.c` files is `gnu11` [COMMUNITY]. To compile for `c90` in Arm Compiler 6, use `-std=c90`.

Arm Compiler 6 generates the following errors and warnings when trying to compile the example `startup_ac5.c` file in `c90` mode:

```
startup_ac5.c:39:22: error: 'main' must return 'int'
__declspec(noreturn) void main (void)
                     ^~~~
                     int
startup_ac5.c:45:9: error: '#pragma import' is an ARM Compiler 5 extension, and is not
supported by ARM Compiler 6 [-Warmcc-pragma-import]
#pragma import (__use_no_semihosting)
        ^
startup_ac5.c:60:7: error: expected '(' after 'asm'
__asm void Vectors(void) {
```

```
      ^
startup_ac5.c:60:6: error: expected ';' after top-level asm block
__asm void Vectors(void) {
     ^
     ;
startup_ac5.c:61:3: error: use of undeclared identifier 'IMPORT'
  IMPORT Undef_Handler
  ^
startup_ac5.c:80:7: error: expected '(' after 'asm'
__asm void Reset_Handler(void) {
      ^
startup_ac5.c:80:6: error: expected ';' after top-level asm block
__asm void Reset_Handler(void) {
     ^
     ;
startup_ac5.c:83:3: error: use of undeclared identifier 'CPSID'
  CPSID   if
  ^
8 errors generated.
```

The following section describes how to modify the source file to fix these errors and warnings.

Modifying the source code for Arm® Compiler 6

You must make the following changes to the source code to compile with `armclang`.

- The return type of function `main` function cannot be **`void`** in standard C. Replace the following line:

  ```
  __declspec(noreturn) void main(void)
  ```

  With:

  ```
  __declspec(noreturn) int main(void)
  ```

- The intrinsic `__enable_irq()` is not supported in Arm Compiler 6. You must replace the intrinsic with an inline assembler equivalent. Replace the following line:

  ```
  __enable_irq();
  ```

  With:

  ```
  __asm("CPSIE i");
  ```

- The `#pragma import` is not supported in Arm Compiler 6. You must replace the pragma with an equivalent directive using inline assembler. Replace the following line:

  ```
  #pragma import(__use_no_semihosting)
  ```

  With:

  ```
  __asm(".global __use_no_semihosting");
  ```

- In certain situations, `armclang` might remove infinite loops that do not have side-effects. You must use the `volatile` keyword to tell `armclang` not to remove such code. Replace the following line:

  ```
  while(1);
  ```

  With:

  ```
  while(1) __asm volatile("");
  ```

Chapter 3
Migrating from armcc to armclang

Compares Arm Compiler 6 command-line options to older versions of Arm Compiler.

It contains the following sections:

3.1 Migration of compiler command-line options from Arm® Compiler 5 to Arm® Compiler 6

Arm Compiler 6 provides many command-line options, including most Clang command-line options and several Arm-specific options.

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page 1-14.

The following table describes the most common Arm Compiler 5 command-line options, and shows the equivalent options for Arm Compiler 6.

More information about command-line options is available:

- The *Arm® Compiler Reference Guide* provides more information about the supported command-line options. The options described are fully supported, unless the level of support is indicated.
- For a full list of Clang command-line options, see the Clang and LLVM documentation.

Table 3-1 Comparison of compiler command-line options in Arm Compiler 5 and Arm Compiler 6

Arm Compiler 5 option	Arm Compiler 6 option	Description
`--allow_fpreg_for_nonfpdata`, `--no_allow_fpreg_for_nonfpdata`	`-mimplicit-float`, `-mno-implicit-float` [COMMUNITY]	Enables or disables the use of VFP and SIMD registers and data transfer instructions for non-VFP and non-SIMD data.
`--apcs=/nointerwork`	No equivalent.	Disables interworking between A32 and T32 code. Interworking is always enabled in Arm Compiler 6.
`--apcs=/ropi` `--apcs=/noropi`	`-fropi` `-fno-ropi`	Enables or disables the generation of Read-Only Position Independent (ROPI) code.
`--apcs=/rwpi` `--apcs=/norwpi`	`-frwpi` `-fno-rwpi`	Enables or disables the generation of Read Write Position Independent (RWPI) code.
`--arm`	`-marm`	Targets the A32 instruction set. The compiler is permitted to generate both A32 and T32 code, but recognizes that A32 code is preferred.
`--arm_only`	`No equivalent.`	Enforces A32 instructions only. The compiler does not generate T32 instructions.
`--asm`	`-save-temps`	Instructs the compiler to generate intermediate assembly files as well as object files.
`--bigend`	`-mbig-endian`	Generates code for big-endian data.
`-c`	`-c`	Performs the compilation step, but not the link step.
`--c90`	`-xc -std=c90`	Enables the compilation of C90 source code. `-xc` is a positional argument and only affects subsequent input files on the command-line. It is also only required if the input files do not have the appropriate file extension.

Table 3-1 Comparison of compiler command-line options in Arm Compiler 5 and Arm Compiler 6 (continued)

Arm Compiler 5 option	Arm Compiler 6 option	Description
`--c90 --gnu`	`-xc -std=gnu90`	Enables the compilation of C90 source code with additional GNU extensions. `-xc` is a positional argument and only affects subsequent input files on the command-line. It is also only required if the input files do not have the appropriate file extension.
`--c99`	`-xc -std=c99`	Enables the compilation of C99 source code. `-xc` is a positional argument and only affects subsequent input files on the command-line. It is also only required if the input files do not have the appropriate file extension.
`--c99 --gnu`	`-xc -std=gnu99`	Enables the compilation of C99 source code with additional GNU extensions. `-xc` is a positional argument and only affects subsequent input files on the command-line. It is also only required if the input files do not have the appropriate file extension.
`--cpp`	`-xc++ -std=c++03`	Enables the compilation of C++03 source code. `-xc++` is a positional argument and only affects subsequent input files on the command-line. It is also only required if the input files do not have the appropriate file extension. The default C++ language standard is different between Arm Compiler 5 and Arm Compiler 6.
`--cpp --gnu`	`-xc++ -std=gnu++03`	Enables the compilation of C++03 source code with additional GNU extensions. `-xc++` is a positional argument and only affects subsequent input files on the command-line. It is also only required if the input files do not have the appropriate file extension. The default C++ language standard is different between Arm Compiler 5 and Arm Compiler 6.
`--cpp11`	`-xc++ -std=c++11`	Enables the compilation of C++11 source code. `-xc++` is a positional argument and only affects subsequent input files on the command-line. It is also only required if the input files do not have the appropriate file extension. The default C++ language standard is different between Arm Compiler 5 and Arm Compiler 6.
`--cpp11 --gnu`	`-xc++ -std=gnu++11`	Enables the compilation of C++11 source code with additional GNU extensions. `-xc++` is a positional argument and only affects subsequent input files on the command-line. It is also only required if the input files do not have the appropriate file extension. The default C++ language standard is different between Arm Compiler 5 and Arm Compiler 6.

Table 3-1 Comparison of compiler command-line options in Arm Compiler 5 and Arm Compiler 6 (continued)

Arm Compiler 5 option	Arm Compiler 6 option	Description
`--cpp_compat`	No equivalent.	Compiles C++ code to maximize binary compatibility.
`--cpu=8-A.32`	`--target=arm-arm-none-eabi -march=armv8-a`	Targets Armv8-A and AArch32 state.
`--cpu 8-A.64`	`--target=aarch64-arm-none-eabi`	Targets Armv8-A and AArch64 state. (Implies `-march=armv8-a` if `-mcpu` is not specified.)
`--cpu=7-A`	`--target=arm-arm-none-eabi -march=armv7-a`	Targets the Armv7-A architecture.
`--cpu=Cortex-M4`	`--target=arm-arm-none-eabi -mcpu=cortex-m4`	Targets the Cortex®-M4 processor.
`--cpu=Cortex-A15`	`--target=arm-arm-none-eabi -mcpu=cortex-a15`	Targets the Cortex-A15 processor.
`-D`	`-D`	Defines a preprocessing macro.
`--depend`	`-MF`	Specifies a filename for the makefile dependency rules.
`--depend_dir`	No equivalent. Use `-MF` to specify each dependency file individually.	Specifies the directory for dependency output files.
`--depend_format=unix_escaped`		Dependency file entries use UNIX-style path separators and escapes spaces with `\`. This is the default in Arm Compiler 6.
`--depend_system_headers`, `--no_depend_system_headers`	No direct equivalent to the standalone command-line option. However, see the Arm Compiler 5 entries in this table for `--md`, `--md --no_depend_system_headers`, and `--mm`.	Enables and disables the output of system include dependency lines when generating makefile dependency information using either the `-M` option or the `--md` option.
`--depend_target`	`-MT`	Changes the target name for the makefile dependency rule.
`--diag_error`	`-Werror`	Turn compiler warnings into errors.
`--diag_style=`*`string`*	No equivalent.	Armclang produces diagnostic messages in the following format: `source-file:line-number:char-number: description [diagnostic-flag]`
`--diag_suppress=`*`foo`*	`-Wno-`*`foo`*	Suppress warning message *`foo`*. The error or warning codes might be different between Arm Compiler 5 and Arm Compiler 6.
`-E`	`-E`	Executes only the preprocessor step.
`--enum_is_int`	`-fno-short-enums`, `-fshort-enums`	Sets the minimum size of an enumeration type. By default Arm Compiler 5 does not set a minimum size. By default Arm Compiler 6 uses `-fno-short-enums` to set the minimum size to 32-bit.

Table 3-1 Comparison of compiler command-line options in Arm Compiler 5 and Arm Compiler 6 (continued)

Arm Compiler 5 option	Arm Compiler 6 option	Description
`--forceline`	No equivalent.	Forces aggressive inlining of functions. Arm Compiler 6 automatically decides whether to inline functions depending on the optimization level.
`--fpmode=std`	`-ffp-mode=std`	Provides IEEE-compliant code with no IEEE exceptions, NaNs, and Infinities. Denormals are sign preserving. This is the default.
`--fpmode=fast`	`-ffp-mode=fast`	Similar to the default behavior, but also performs aggressive floating-point optimizations and therefore it is not IEEE-compliant.
`--fpmode=ieee_full`	`-ffp-mode=full`	Provides full IEEE support, including exceptions.
`--fpmode=ieee_fixed` `--fpmode=ieee_no_fenv`	There are no supported equivalent options.	There might be community features that provide these IEEE floating-point modes.
`--fpu` For example, `--fpu=fpv5_d16`	`-mfpu` For example, `-mfpu=fpv5-d16`	Specifies the target FPU architecture. **Note** `--fpu=none` checks the source code for floating-point operations, and if any are found it produces an error. `-mfpu=none` prevents the compiler from using hardware-based floating-point functions. If the compiler encounters floating-point types in the source code, it uses software-based floating-point library functions. The option values might be different. For example `fpv5_d16` in Arm Compiler 5 is equivalent to `fpv5-d16` in Arm Compiler 6, and targets the FPv5-D16 floating-point extension.
`-I`	`-I`	Adds the specified directories to the list of places that are searched to find included files.
`--global_reg=`*`reg_name`*	`-ffixed-r`*`N`*	Prevents the compiler from using the specified core register, unless the use is required for Arm ABI compliance. In Arm Compiler 5, *`reg_name`* is an integer starting from `1` to `8`, which maps to registers R4 to R11. In Arm Compiler 6, *`N`* is an integer starting from `5` to `11`, which maps to registers R5 to R11.
`--ignore_missing_headers`	`-MG`	Prints dependency lines for header files even if the header files are missing.
`--inline`	Default at `-O2` and `-O3`.	There is no equivalent of the `--inline` option. Arm Compiler 6 automatically decides whether to inline functions at optimization levels `-O2` and `-O3`.
`-J`	`-isystem`	Adds the specified directories to the list of places that are searched to find included system header files.
`-L`	`-Xlinker`	Specifies command-line options to pass to the linker when a link step is being performed after compilation.

Table 3-1 Comparison of compiler command-line options in Arm Compiler 5 and Arm Compiler 6 (continued)

Arm Compiler 5 option	Arm Compiler 6 option	Description
`--library_interface=armcc`	This is the default.	Arm Compiler 6 by default uses the Arm standard C library.
`--library_interface=lib` Where *`lib`* is one of: - `aeabi_clib` - `aeabi_clib90` - `aeabi_clib99`	`-nostdlib -nostdlibinc -fno-builtin`	Specifies that the compiler output works with any ISO C library compliant with the Arm Embedded Application Binary Interface (AEABI).
`--library_interface=lib` Where *`lib`* is not one of: - `aeabi_clib` - `aeabi_clib90` - `aeabi_clib99` - `armcc`	No equivalent.	Arm Compiler 6 assumes the use of an AEABI compliant library.
`--licretry`	No equivalent.	There is no equivalent of the `--licretry` option. The Arm Compiler 6 tools automatically retry failed attempts to obtain a license.
`--list_macros`	`-E -dM`	List all the macros that are defined at the end of the translation unit, including the predefined macros.
`--littleend`	`-mlittle-endian`	Generates code for little-endian data.
`--lower_ropi`, `--no_lower_ropi`	`-fropi-lowering`, `-fno-ropi-lowering`	Enables or disables less restrictive C when generating Read-Only Position Independent (ROPI) code. **Note** In Arm Compiler 5, when`--acps=/ropi` is specified, `--lower_ropi` is not switched on by default. In Arm Compiler 6, when `-fropi` is specified, `-fropi-lowering` is switched on by default.
`--lower_rwpi`, `--no_lower_rwpi`	`-frwpi-lowering`, `-fno-rwpi-lowering`	Enables or disables less restrictive C when generating Read Write Position Independent (RWPI) code.
`-M`	`-M`	Instructs the compiler to produce a list of makefile dependency lines suitable for use by a make utility.
`--md`	`-MD`	Creates makefile dependency files, including the system header files. In Arm Compiler 5, this is equivalent to `--md --depend_system_headers`.
`--md --no_depend_system_headers`	`-MMD`	Creates makefile dependency files, without the system header files.
`--mm`	`-MM`	Creates a single makefile dependency file, without the system header files. In Arm Compiler 5, this is equivalent to `-M --no_depend_system_headers`.
`--multifile`, `--no_multifile`	No direct equivalent. However, see *Optimizing across modules with Link-Time Optimization* in the *Arm® Compiler User Guide*.	Enables and disables optimizations between multiple source files.

Table 3-1 Comparison of compiler command-line options in Arm Compiler 5 and Arm Compiler 6 (continued)

Arm Compiler 5 option	Arm Compiler 6 option	Description
`--no_comment_section`	`-fno-ident`	Removes the `.comment` section from object files.
`--no_exceptions`	`-fno-exceptions`	Disables the generation of code needed to support C++ exceptions. **Note** For C++ code, Arm Compiler 6 defaults to `-fexceptions`. As a result, there might be a large increase in the code size. If you use `-fno_exceptions`, then the code size is in the range of that created with Arm Compiler 5.
`--no_hide_all`	`-fvisibility=default`	Sets the default visibility of ELF symbols to the specified option, unless overridden in the source with the `__attribute__((visibility("`*`visibility_type`*`")))` attribute. The default is `-fvisibility=hidden`. **Note** The behavior of `armclang -fvisibility=hidden` is different from that of the `armcc --hide-all` option. With `armclang -fvisibility=hidden`, **`extern`** declarations are visible, and all other symbols are hidden. With `armcc --hide-all`, all symbols are hidden.
`--no_protect_stack`	`-fno-stack-protector`	Explicitly disables stack protection. See *3.2 Arm® Compiler 5 and Arm® Compiler 6 stack protection behavior* on page 3-40 for more information.
`-no_rtti`, `-no_rtti_data`	`-fno-rtti` [ALPHA]	C++ only. `-fno-rtti` [ALPHA] disables the generation of code that is needed to support Run Time Type Information (RTTI) features.
`-o`	`-o`	Specifies the name of the output file.
`-O`*`num`*	`-O`*`num`*	Specifies the level of optimization to be used when compiling source files. The default for Arm Compiler 5 is `-O2`. The default for Arm Compiler 6 is `-O0`. For Arm Compiler 6, Arm recommends `-O1` rather than `-O0` for best trade-off between debug view, code size, and performance. For more information, see *Optimization differences*.
`-Ospace`	`-Oz / -Os`	Performs optimizations to reduce image size at the expense of a possible increase in execution time.
`-Otime`	This is the default.	Performs optimizations to reduce execution time at the expense of a possible increase in image size. There is no equivalent of the `-Otime` option. Arm Compiler 6 optimizes for execution time by default, unless you specify the `-Os` or `-Oz` options.
`--phony_targets`	`-MP`	Emits dummy makefile rules.

Table 3-1 Comparison of compiler command-line options in Arm Compiler 5 and Arm Compiler 6 (continued)

Arm Compiler 5 option	Arm Compiler 6 option	Description
`--preinclude`	`-include`	Include the source code of a specified file at the beginning of the compilation.
`--protect_stack`	`-fstack-protector`, `-fstack-protector-strong`	Enables stack protection on vulnerable functions. See *3.2 Arm® Compiler 5 and Arm® Compiler 6 stack protection behavior* on page 3-40 for more information.
`--protect_stack_all`	`-fstack-protector-all`	Enables stack protection on all functions. See *3.2 Arm® Compiler 5 and Arm® Compiler 6 stack protection behavior* on page 3-40 for more information.
`--relaxed_ref_def`	`-fcommon`	Places zero-initialized definitions in a common block.
`--retain`	`-O`	The optimization level to use for the best code coverage might depend on your source code. In Arm Compiler 5 the `--retain` option disables specific optimizations by name. There is no direct equivalent of this for Arm Compiler 6. Instead you will need to select the optimization level which best suits your needs. See *-O* in the *Arm® Compiler Reference Guide* for more information.
`-S`	`-S`	Outputs the disassembly of the machine code that the compiler generates. The output from this option differs between releases. Arm Compiler 5 produces output with `armasm` syntax while Arm Compiler 6 produces output with GNU syntax.
`--show_cmdline`	`-v`	Shows how the compiler processes the command-line. The commands are shown normalized, and the contents of any via files are expanded.
`--split_ldm`	`-fno-ldm-stm`	Disables the generation of `LDM` and `STM` instructions. Note that while the `armcc --split_ldm` option limits the size of generated LDM/STM instructions, the `armclang -fno-ldm-stm` option disables the generation of `LDM` and `STM` instructions altogether.
`--split_sections`	`-ffunction-sections`	Generates one ELF section for each function in the source file. In Arm Compiler 6, `-ffunction-sections` is the default. Therefore, the merging of identical constants cannot be done by `armclang`. Instead, the merging is done by `armlink`. See *Merging identical constants* in the *Arm® Compiler Reference Guide* for more information.
`--strict`	`-pedantic-errors`	Generate errors if code violates strict ISO C and ISO C++.
`--strict_warnings`	`-pedantic`	Generate warnings if code violates strict ISO C and ISO C++.
`--thumb`	`-mthumb`	Targets the T32 instruction set.
`--no_unaligned_access`, `--unaligned_access`	`-mno-unaligned-access`, `-munaligned-access`	Enables or disables unaligned accesses to data on Arm processors.

Table 3-1 Comparison of compiler command-line options in Arm Compiler 5 and Arm Compiler 6 (continued)

Arm Compiler 5 option	Arm Compiler 6 option	Description
`--use_frame_pointer`, `--no_use_frame_pointer`	`-fno-omit-frame-pointer`, `-fomit-frame-pointer`	Controls whether a register is reserved for storing the stack frame pointer.
`--vectorize` `--no_vectorize`	`-fvectorize` `-fno-vectorize`	Enables or disables the generation of Advanced SIMD vector instructions directly from C or C++ code.
`--via`	`@file`	Reads an additional list of compiler options from a file.
`--vla`	No equivalent.	Support for variable length arrays. Arm Compiler 6 automatically supports variable length arrays in accordance with the language standard.
`--vsn`	`--version`	Displays version information and license details. In Arm Compiler 6 you can also use `--vsn`.
`--wchar16`, `--wchar32`	`-fshort-wchar`, `-fno-short-wchar`	Sets the size of **`wchar_t`** type. The default for Arm Compiler 5 is `--wchar16`. The default for Arm Compiler 6 is `-fno-short-wchar`.

Related information

armclang Command-line Options

Compiler-specific Function, Variable, and Type Attributes

The LLVM Compiler Infrastructure Project

3.2 Arm® Compiler 5 and Arm® Compiler 6 stack protection behavior

You can see which functions are protected and compare Arm Compiler 5 protection with Arm Compiler 6 protection after migration.

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page 1-14.

The behavior of `armclang -fstack-protector` and `armclang -fstack-protector-strong` is different from the behavior of the `armcc --protect_stack` option:

- With `armcc --protect_stack`, a function is considered vulnerable if it contains a **`char`** or **`wchar_t`** array of any size.
- With `armclang -fstack-protector`, a function is considered vulnerable if it contains at least one of the following:
  - A character array larger than 8 bytes.
  - An 8-bit integer array larger than 8 bytes.
  - A call to `alloca()` with either a variable size or a constant size bigger than 8 bytes.
- With `armclang -fstack-protector-strong`, a function is considered vulnerable if it contains:
  - An array of any size and type.
  - A call to `alloca()`.
  - A local variable that has its address taken.

  Arm recommends the use of `-fstack-protector-strong`.

Note

When using Arm Compiler 5, the value of the variable `__stack_chk_guard` could change during the life of the program. With Arm Compiler 6, a suitable implementation might set this variable to a random value when the program is loaded, before the first protected function is entered. The value must then remain unchanged during the life of the program.

Example

1. Create the file `test.c` containing the following code:

```
// test.c

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

void *__stack_chk_guard = (void *)0xdeadbeef;

void __stack_chk_fail(void) {
  printf("Stack smashing detected.\n");
  exit(1);
}

static void copy(const char *p) {
  char buf[8];
  strcpy(buf, p);
  printf("Copied: %s\n", buf);
}

int main(void) {
  const char *t = "Hello World!";
  copy(t);
  printf("%s\n", t);
```

```
  return 0;
}
```

2. For Arm Compiler 5, search for branches to the `__stack_chk_fail()` function in the output from the `fromelf -c` command. The functions containing such branches are protected.

```
armcc -c --cpu=7-A --protect_stack test.c -o test.o
fromelf -c test.o
...
    copy
        0x00000010:    e92d403e    >@-.    PUSH     {r1-r5,lr}
        0x00000014:    e1a04000    .@..    MOV      r4,r0
        0x00000018:    e59f0070    p...    LDR      r0,[pc,#112] ; [__stack_chk_guard = 
0x90] = 0
        0x0000001c:    e5905000    .P..    LDR      r5,[r0,#0]
        0x00000020:    e58d5008    .P..    STR      r5,[sp,#8]
        0x00000024:    e1a01004    ....    MOV      r1,r4
        0x00000028:    e1a0000d    ....    MOV      r0,sp
        0x0000002c:    ebfffffe    ....    BL       strcpy
        0x00000030:    e1a0100d    ....    MOV      r1,sp
        0x00000034:    e28f0058    X...    ADR      r0,{pc}+0x60 ; 0x94
        0x00000038:    ebfffffe    ....    BL       __2printf
        0x0000003c:    e59d0008    ....    LDR      r0,[sp,#8]
        0x00000040:    e1500005    ..P.    CMP      r0,r5
        0x00000044:    0a000000    ....    BEQ      {pc}+0x8 ; 0x4c
        0x00000048:    ebfffffe    ....    BL       __stack_chk_fail ; 0x0 Section #1
        0x0000004c:    e8bd803e    >...    POP      {r1-r5,pc}
...
```

3. For Arm Compiler 6, use the `armclang` [COMMUNITY] `-Rpass` remark option.

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -O0 -Rpass=stack-protector test.c
test.c:14:13: remark: Stack protection applied to function copy due to a stack allocated
buffer or struct containing a
      buffer [-Rpass=stack-protector]
static void copy(const char *p) {
            ^
```

Note

You can also use the `fromelf -c` command and search the output for functions containing branches to the `__stack_chk_fail()` function.

Related information

Reference Guide: -Rpass

Reference Guide: -fstack-protector, -fstack-protector-strong, -fstack-protector-all, -fno-stack-protector

3.3 Command-line options for preprocessing assembly source code

The functionality of the `--cpreproc` and `--cpreproc_opts` command-line options in the version of `armasm` supplied with Arm Compiler 6 is different from the options used in earlier versions of `armasm` to preprocess assembly source code.

If you are using `armasm` to assemble source code that requires the use of the preprocessor, you must use both the `--cpreproc` and `--cpreproc_opts` options together. Also:

- As a minimum, you must include the `armclang` options `--target` and either `-mcpu` or `-march` in `--cpreproc_opts`.
- The input assembly source must have an upper-case extension `.S`.

If you have existing source files, which require preprocessing, and that have the lower-case extension `.s`, then to avoid having to rename the files:

1. Perform the preprocessing step separately using the `armclang -x assembler-with-cpp` option.
2. Assemble the preprocessed file without using the `--cpreproc` and `--cpreproc_opts` options.

Example using armclang -x

This example shows the use of the armclang `-x` option.

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a -x assembler-with-cpp -E test.s -o
test_preproc.s
armasm --cpu=8-A.64 test_preproc.s
```

Example using armasm --cpreproc_opts

The options to the preprocessor in this example are `--cpreproc_opts=--target=arm-arm-none-eabi,-mcpu=cortex-a9,-D,DEF1,-D,DEF2`.

```
armasm --cpu=cortex-a9 --cpreproc --cpreproc_opts=--target=arm-arm-none-eabi,-mcpu=cortex-
a9,-D,DEF1,-D,DEF2 -I /path/to/includes1 -I /path/to/includes2 input.S
```

Note

Ensure that you specify compatible architectures in the `armclang` options `--target`, `-mcpu` or `-march`, and the `armasm` `--cpu` option.

Related information

--cpreproc assembler option

--cpreproc_opts assembler option

Mandatory armclang options

-march armclang option

-mcpu armclang option

--target armclang option

-x armclang option

Preprocessing assembly code

3.4 Inline assembly with Arm® Compiler 6

Inline assembly in Arm Compiler 6 must be written in GNU assembly syntax. Inline assembly in Arm Compiler 5 is written in `armasm` syntax. If you have inline assembly written in `armasm` syntax, you must modify the `armasm` syntax assembly to use GNU assembly syntax.

In Arm Compiler 5:

- You can use C variable names directly inside inline assembly statements.
- You do not have direct access to physical registers. You must use C or C++ variables names as operands, and the compiler maps them to physical register. You must set the value of these variables before you read them within an inline assembly statement.
- If you use register names in inline assembly code, they are treated as C or C++ variables. They do not necessarily relate to the physical register of the same name. If the register name is not declared as a C or C++ variable, the compiler generates a warning.

In Arm Compiler 6:

- You cannot use C or C++ variable names directly inside inline assembly statements. You can map the physical registers to C or C++ variable names using operand mapping and constraints.
- You have direct access to physical registers. There is no need to set the value of the registers before you read them within inline assembly statements.
- If you use register names in inline assembly code, they are the physical register of the same name.

In Arm Compiler 6 you cannot use C variable names directly within inline assembly. However, the GNU assembly syntax in Arm Compiler 6 provides a way for mapping input and output operands to C variable names.

Arm Compiler 5 optimizes inline assembly, but Arm Compiler 6 emits it exactly as written.

Note

While Arm Compiler 6 does not attempt to optimize the inline assembly instructions, it can remove a block of code containing inline assembly during optimization. The compiler is unaware of the content of the assembly, so might in some cases remove the block while attempting to remove unused code.

The `volatile` qualifier disables certain compiler optimizations, which might otherwise lead to the compiler removing the code block. The `volatile` qualifier is OPTIONAL. However, consider using it around your assembly code blocks to ensure the compiler does not remove them when compiling at any optimization level other than `-O0`.

See the documentation of the `volatile` keyword in the *Arm Compiler 6 User Guide* for details.

For more information on writing inline assembly using `__asm` in `armclang`, see *__asm*.

For more information on GNU assembly syntax, see *5.2 Overview of differences between armasm and GNU syntax assembly code* on page 5-81.

Inline assembly example in Arm® Compiler 5

The following example shows inline assembly code in Arm Compiler 5:

```
foo.c:

int add(int i, int j)
{
    int res;
    __asm
    (
      "ADD res, i, j \t\n"
      "SUB res, i, res \t\n"
    );
    return res;
}
```

The following example shows an alternative syntax for inline assembly code in Arm Compiler 5:

```
foo.c:

int add(int i, int j)
{
    int res;
    __asm
    {
        ADD   res, i, j
        SUB   res, i, res
    }
    return res;
}
```

Compile `foo.c` using `armcc`:

```
armcc foo.c -c -S -o foo.s
```

Arm Compiler 5 converts the example inline assembly code to:

```
foo.s:

add PROC
        ADD r1,r0,r1
        SUB r0,r0,r1
        BX  lr
        ENDP
```

Inline assembly example in Arm® Compiler 6

The example below shows the equivalent inline assembly code in Arm Compiler 6.

```
foo.c:

int add(int i, int j)
{
  int res = 0;
  __asm
  (
    "ADD %[result], %[input_i], %[input_j] \t\n"
    "SUB %[result], %[input_i], %[result]  \t\n"
    : [result] "=&r" (res)
    : [input_i] "r" (i), [input_j] "r" (j)
  );
  return res;
}
```

Compile `foo.c` using `armclang` with optimization level `-O1`:

```
armclang foo.c --target=arm-arm-none-eabi -march=armv8-a -O1 -c -S -o foo.s
```

Arm Compiler 6 converts the example inline assembly code to:

```
foo.s:

add:
        .fnstart
@ BB#0:
        @APP
        add r2,r0,r1
        sub r2,r0,r2
        @NO_APP
        mov r0,r2
        bx  lr
```

Note

Arm Compiler 6 supports inline assembly using the `__asm` or `asm` keywords. However, the `asm` keyword is accepted only when:

- Used within C++ language source files.
- Used within C language source files without strict ISO C Standard compliance. For example, `asm` is accepted when using `-std=gnu11`.

Related information

armclang Inline Assembler

How to Use Inline Assembly Language in C Code

Constraints for asm Operands

Constraint Modifier Characters

3.5 Migrating architecture and processor names for command-line options

There are minor differences between the architecture and processor names that Arm Compiler 6 recognizes, and the names that Arm Compiler 5 recognizes. Within Arm Compiler 6, there are differences in the architecture and processor names that `armclang` recognizes and the names that `armasm`, `armlink`, and `fromelf` recognize. This topic shows the differences in the architecture and processor names for the different tools in Arm Compiler 5 and Arm Compiler 6.

The tables show the documented `--cpu` options in Arm Compiler 5 and their corresponding options for migrating your Arm Compiler 5 command-line options to Arm Compiler 6.

Note

The tables assume the default floating-point unit derived from the `--cpu` option in Arm Compiler 5. However, in Arm Compiler 6, `armclang` selects different defaults for floating-point unit (VFP) and Advanced SIMD. Therefore, the tables also show how to use the `armclang` `-mfloat-abi` and `-mfpu` options to be compatible with the default floating-point unit in Arm Compiler 5. The tables do not provide an exhaustive list.

Table 3-2 Architecture selection in Arm Compiler 5 and Arm Compiler 6

`armcc`, `armlink`, `armasm`, and `fromelf` option in Arm Compiler 5	`armclang` option in Arm Compiler 6	`armlink`, `armasm`, and `fromelf` option in Arm Compiler 6	Architecture description
`--cpu=4`	Not supported	Not supported	Armv4
`--cpu=4T`	Not supported	Not supported	Armv4T
`--cpu=5T`	Not supported	Not supported	Armv5T
`--cpu=5TE`	Not supported	Not supported	Armv5TE
`--cpu=5TEJ`	Not supported	Not supported	Armv5TEJ
`--cpu=6`	Not supported	Not supported	Generic Armv6
`--cpu=6-K`	Not supported	Not supported	Armv6-K
`--cpu=6-Z`	Not supported	Not supported	Armv6-Z
`--cpu=6T2`	Not supported	Not supported	Armv6T2
`--cpu=6-M`	`--target=arm-arm-none-eabi -march=armv6-m`	`--cpu=6S-M`	Armv6-M
`--cpu=6S-M`	`--target=arm-arm-none-eabi -march=armv6s-m`	`--cpu=6S-M`	Armv6S-M

Table 3-2 Architecture selection in Arm Compiler 5 and Arm Compiler 6 (continued)

`armcc`, `armlink`, `armasm`, and `fromelf` option in Arm Compiler 5	`armclang` option in Arm Compiler 6	`armlink`, `armasm`, and `fromelf` option in Arm Compiler 6	Architecture description
`--cpu=7-A` `--cpu=7-A.security`	`--target=arm-arm-none-eabi -march=armv7-a -mfloat-abi=soft`	`--cpu=7-A.security`	Armv7-A without VFP and Advanced SIMD. In Arm Compiler 5, security extension is not enabled with `--cpu=7-A` but is enabled with `--cpu=7-A.security`. In Arm Compiler 6, `armclang` always enables the Armv7-A TrustZone security extension with `-march=armv7-a`. However, `armclang` does not generate an `SMC` instruction unless you specify it with an intrinsic or inline assembly.
`--cpu=7-R`	`--target=arm-arm-none-eabi -march=armv7-r -mfloat-abi=soft`	`--cpu=7-R`	Armv7-R without VFP and Advanced SIMD
`--cpu=7-M`	`--target=arm-arm-none-eabi -march=armv7-m`	`--cpu=7-M`	Armv7-M
`--cpu=7E-M`	`--target=arm-arm-none-eabi -march=armv7e-m -mfloat-abi=soft`	`--cpu=7E-M`	Armv7E-M

Table 3-3 Processor selection in Arm Compiler 5 and Arm Compiler 6

`armcc`, `armlink`, `armasm`, and `fromelf` option in Arm Compiler 5	`armclang` option in Arm Compiler 6	`armlink`, `armasm`, and `fromelf` option in Arm Compiler 6	Description
`--cpu=Cortex-A5`	`--target=arm-arm-none-eabi -mcpu=cortex-a5 -mfloat-abi=soft`	`--cpu=Cortex-A5.no_neon.no_vfp`	Cortex-A5 without Advanced SIMD and VFP
`--cpu=Cortex-A5.neon`	`--target=arm-arm-none-eabi -mcpu=cortex-a5 -mfloat-abi=hard`	`--cpu=Cortex-A5`	Cortex-A5 with Advanced SIMD and VFP
`--cpu=Cortex-A5.vfp`	`--target=arm-arm-none-eabi -mcpu=cortex-a5 -mfloat-abi=hard -mfpu=vfpv4-d16`	`--cpu=Cortex-A5.no_neon`	Cortex-A5 with VFP, without Advanced SIMD
`--cpu=Cortex-A7`	`--target=arm-arm-none-eabi -mcpu=cortex-a7 -mfloat-abi=hard`	`--cpu=Cortex-A7`	Cortex-A7 with Advanced SIMD and VFP
`--cpu=Cortex-A7.no_neon.no_vfp`	`--target=arm-arm-none-eabi -mcpu=cortex-a7 -mfloat-abi=soft`	`--cpu=Cortex-A7.no_neon.no_vfp`	Cortex-A7 without Advanced SIMD and VFP

Table 3-3 Processor selection in Arm Compiler 5 and Arm Compiler 6 (continued)

`armcc`, `armlink`, `armasm`, and `fromelf` option in Arm Compiler 5	`armclang` option in Arm Compiler 6	`armlink`, `armasm`, and `fromelf` option in Arm Compiler 6	Description
`--cpu=Cortex-A7.no_neon`	`--target=arm-arm-none-eabi -mcpu=cortex-a7 -mfloat-abi=hard -mfpu=vfpv4-d16`	`--cpu=Cortex-A7.no_neon`	Cortex-A7 with VFP, without Advanced SIMD
`--cpu=Cortex-A8`	`--target=arm-arm-none-eabi -mcpu=cortex-a8 -mfloat-abi=hard`	`--cpu=Cortex-A8`	Cortex-A8 with VFP and Advanced SIMD
`--cpu=Cortex-A8.no_neon`	`--target=arm-arm-none-eabi -mcpu=cortex-a8 -mfloat-abi=soft`	`--cpu=Cortex-A8.no_neon`	Cortex-A8 without Advanced SIMD and VFP
`--cpu=Cortex-A9`	`--target=arm-arm-none-eabi -mcpu=cortex-a9 -mfloat-abi=hard`	`--cpu=Cortex-A9`	Cortex-A9 with Advanced SIMD and VFP
`--cpu=Cortex-A9.no_neon.no_vfp`	`--target=arm-arm-none-eabi -mcpu=cortex-a9 -mfloat-abi=soft`	`--cpu=Cortex-A9.no_neon.no_vfp`	Cortex-A9 without Advanced SIMD and VFP
`--cpu=Cortex-A9.no_neon`	`--target=arm-arm-none-eabi -mcpu=cortex-a9 -mfloat-abi=hard -mfpu=vfpv3-d16-fp16`	`--cpu=Cortex-A9.no_neon`	Cortex-A9 with VFP but without Advanced SIMD
`--cpu=Cortex-A12`	`--target=arm-arm-none-eabi -mcpu=cortex-a12 -mfloat-abi=hard`	`--cpu=Cortex-A12`	Cortex-A12 with Advanced SIMD and VFP
`--cpu=Cortex-A12.no_neon.no_vfp`	`--target=arm-arm-none-eabi -mcpu=cortex-a12 -mfloat-abi=soft`	`--cpu=Cortex-A12.no_neon.no_vfp`	Cortex-A12 without Advanced SIMD and VFP
`--cpu=Cortex-A15`	`--target=arm-arm-none-eabi -mcpu=cortex-a15 -mfloat-abi=hard`	`--cpu=Cortex-A15`	Cortex-A15 with Advanced SIMD and VFP
`--cpu=Cortex-A15.no_neon`	`--target=arm-arm-none-eabi -mcpu=cortex-a15 -mfloat-abi=hard -mfpu=vfpv4-d16`	`--cpu=Cortex-A15.no_neon`	Cortex-A15 with VFP, without Advanced SIMD
`--cpu=Cortex-A15.no_neon.no_vfp`	`--target=arm-arm-none-eabi -mcpu=cortex-a15 -mfloat-abi=soft`	`--cpu=Cortex-A15.no_neon.no_vfp`	Cortex-A15 without Advanced SIMD and VFP
`--cpu=Cortex-A17`	`--target=arm-arm-none-eabi -mcpu=cortex-a17 -mfloat-abi=hard`	`--cpu=Cortex-A17`	Cortex-A17 with Advanced SIMD and VFP
`--cpu=Cortex-A17.no_neon.no_vfp`	`--target=arm-arm-none-eabi -mcpu=cortex-a17 -mfloat-abi=soft`	`--cpu=Cortex-A17.no_neon.no_vfp`	Cortex-A17 without Advanced SIMD and VFP

Table 3-3 Processor selection in Arm Compiler 5 and Arm Compiler 6 (continued)

`armcc`, `armlink`, `armasm`, and `fromelf` option in Arm Compiler 5	`armclang` option in Arm Compiler 6	`armlink`, `armasm`, and `fromelf` option in Arm Compiler 6	Description
`--cpu=Cortex-R4`	`--target=arm-arm-none-eabi -mcpu=cortex-r4`	`--cpu=Cortex-R4`	Cortex-R4 without VFP
`--cpu=Cortex-R4F`	`--target=arm-arm-none-eabi -mcpu=cortex-r4f -mfloat-abi=hard`	`--cpu=Cortex-R4F`	Cortex-R4 with VFP
`--cpu=Cortex-R5`	`--target=arm-arm-none-eabi -mcpu=cortex-r5 -mfloat-abi=soft`	`--cpu=Cortex-R5.no_vfp`	Cortex-R5 without VFP
`--cpu=Cortex-R5F`	`--target=arm-arm-none-eabi -mcpu=cortex-r5 -mfloat-abi=hard`	`--cpu=Cortex-R5`	Cortex-R5 with double precision VFP
`--cpu=Cortex-R5F-rev1.sp`	`--target=arm-arm-none-eabi -mcpu=cortex-r5 -mfloat-abi=hard -mfpu=vfpv3xd`	`--cpu=Cortex-R5.sp`	Cortex-R5 with single precision VFP
`--cpu=Cortex-R7`	`--target=arm-arm-none-eabi -mcpu=cortex-r7 -mfloat-abi=hard`	`--cpu=Cortex-R7`	Cortex-R7 with VFP
`--cpu=Cortex-R7.no_vfp`	`--target=arm-arm-none-eabi -mcpu=cortex-r7 -mfloat-abi=soft`	`--cpu=Cortex-R7.no_vfp`	Cortex-R7 without VFP
`--cpu=Cortex-R8`	`--target=arm-arm-none-eabi -mcpu=cortex-r8 -mfloat-abi=hard`	`--cpu=Cortex-R8`	Cortex-R8 with VFP
`--cpu=Cortex-R8.no_vfp`	`--target=arm-arm-none-eabi -mcpu=cortex-r8 -mfloat-abi=soft`	`--cpu=Cortex-R8.no_vfp`	Cortex-R8 without VFP
`--cpu=Cortex-M0`	`--target=arm-arm-none-eabi -mcpu=cortex-m0`	`--cpu=Cortex-M0`	Cortex-M0
`--cpu=Cortex-M0plus`	`--target=arm-arm-none-eabi -mcpu=cortex-m0plus`	`--cpu=Cortex-M0plus`	Cortex-M0+
`--cpu=Cortex-M1`	`--target=arm-arm-none-eabi -mcpu=cortex-m1`	`--cpu=Cortex-M1`	Cortex-M1
`--cpu=Cortex-M3`	`--target=arm-arm-none-eabi -mcpu=cortex-m3`	`--cpu=Cortex-M3`	Cortex-M3
`--cpu=Cortex-M4`	`--target=arm-arm-none-eabi -mcpu=cortex-m4 -mfloat-abi=soft`	`--cpu=Cortex-M4.no_fp`	Cortex-M4 without VFP
`--cpu=Cortex-M4.fp`	`--target=arm-arm-none-eabi -mcpu=cortex-m4 -mfloat-abi=hard`	`--cpu=Cortex-M4`	Cortex-M4 with VFP

Table 3-3 Processor selection in Arm Compiler 5 and Arm Compiler 6 (continued)

`armcc`, `armlink`, `armasm`, and `fromelf` option in Arm Compiler 5	`armclang` option in Arm Compiler 6	`armlink`, `armasm`, and `fromelf` option in Arm Compiler 6	Description
`--cpu=Cortex-M7`	`--target=arm-arm-none-eabi -mcpu=cortex-m7 -mfloat-abi=soft`	`--cpu=Cortex-M7.no_fp`	Cortex-M7 without VFP
`--cpu=Cortex-M7.fp.dp`	`--target=arm-arm-none-eabi -mcpu=cortex-m7 -mfloat-abi=hard`	`--cpu=Cortex-M7`	Cortex-M7 with double precision VFP
`--cpu=Cortex-M7.fp.sp`	`--target=arm-arm-none-eabi -mcpu=cortex-m7 -mfloat-abi=hard -mfpu=fpv5-sp-d16`	`--cpu=Cortex-M7.fp.sp`	Cortex-M7 with single precision VFP

Enabling or disabling architectural features in Arm® Compiler 6

Arm Compiler 6, by default, automatically enables or disables certain architectural features such as the floating-point unit, Advanced SIMD, and Cryptographic extensions depending on the specified architecture or processor. For a list of architectural features, see `-mcpu` in the *Arm® Compiler Reference Guide*. You can override the defaults using other options.

For `armclang`:

- For AArch64 targets, you must use either `-march` or `-mcpu` to specify the architecture or processor and the required architectural features. You can use `+[no]feature` with `-march` or `-mcpu` to override any architectural feature.
- For AArch32 targets, you must use either `-march` or `-mcpu` to specify the architecture or processor and the required architectural features. You can use `-mfloat-abi` to override floating-point linkage. You can use `-mfpu` to override floating-point unit, Advanced SIMD, and Cryptographic extensions. You can use `+[no]feature` with `-march` or `-mcpu` to override certain other architectural features.

For `armasm`, `armlink`, and `fromelf`, you must use the `--cpu` option to specify the architecture or processor and the required architectural features. You can use `--fpu` to override the floating-point unit and floating-point linkage. The `--cpu` option is not mandatory for `armlink` and `fromelf`, but is mandatory for `armasm`.

Note

- In Arm Compiler 5, if you use the `armcc --fpu=none` option, the compiler generates an error if it detects floating-point code. This behavior is different in Arm Compiler 6. If you use the `armclang -mfpu=none` option, the compiler automatically uses software floating-point libraries if it detects any floating-point code. You cannot use the `armlink --fpu=none` option to link object files created using `armclang`.
- To link object files created using the `armclang -mfpu=none` option, you must set `armlink --fpu` to an option that supports software floating-point linkage, for example `--fpu=SoftVFP`, rather than using `--fpu=none`.

Related information

armclang -mcpu option

armclang -march option

armclang -mfloat-abi option

armclang --mfpu option

armclang --target option

armlink --cpu option

armlink --fpu option
fromelf --cpu option
fromelf --fpu option
armasm --cpu option
armasm --fpu option

3.6 Preprocessing a scatter file when linking with armlink

Preprocessing a scatter file when linking with `armlink` in Arm Compiler 6 requires extra options.

The following shows the required change to the first line of the scatter file:

Arm Compiler 5

```
#!armcc -E
```

Arm Compiler 6

```
#!armclang -E --target=arm-arm-none-eabi -mcpu=cortex-m7 -xc
```

The mandatory option `--target` specifies the target state, either AArch32 state, as shown in this example, or AArch64 state. See *--target*.

The option `-mcpu` specifies a processor, Cortex-M7 in this example. Alternatively, you can use `-march` to specify an architecture. See *-mcpu* or *-march*.

The option `-x` specifies the source language. See *-x*.

The option `-E` makes `armclang` only execute the preprocessor step. See *-E*.

3.7 Migrating predefined macros

The functionality of the Arm Compiler 5 predefined macro `__MODULE__` is provded by the `__FILE_NAME__` macro in Arm Compiler 6.

Related information

Predefined macros

Chapter 4
Compiler Source Code Compatibility

Provides details of source code compatibility between Arm Compiler 6 and older `armcc` compiler versions.

It contains the following sections:

4.1 Language extension compatibility: keywords

Arm Compiler 6 supports some keywords that are supported in Arm Compiler 5.

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page 1-14.

The following table lists some of the commonly used keywords that Arm Compiler 5 supports and shows whether Arm Compiler 6 supports them using `__attribute__`. Replace any instances of these keywords in your code with the recommended alternative where available or use inline assembly instructions.

Note

This table is not an exhaustive list of all keywords.

Table 4-1 Keyword language extensions in Arm Compiler 5 and Arm Compiler 6

Keyword supported by Arm Compiler 5	Recommended Arm Compiler 6 keyword or alternative
`__align(x)`	`__attribute__((aligned(x)))`
`__alignof__`	`__alignof__`
`__ALIGNOF__`	`__alignof__`
Embedded assembly using `__asm`	Arm Compiler 6 does not support the `__asm` keyword on function definitions and declarations for embedded assembly. Instead, you can write embedded assembly using the `__attribute__((naked))` function attribute. See *`__attribute__((naked))`*.
`__const`	`__attribute__((const))` **Note** Older versions of `armcc` supported the `__const` keyword. The equivalent for this keyword in Arm Compiler 5 and Arm Compiler 6 is `__attribute__((const))`.
`__attribute__((const))`	`__attribute__((const))`
`__forceinline`	• For C90, use `__inline__` and `__attribute__((always_inline))` • For other source languages, use `inline` and `__attribute__((always_inline))` See *`__attribute__((always_inline))`*.
`__global_reg(N)`	Use the `register` and `__asm` keywords for global named register variables using core registers. For example: `register int Reg5 __asm("r5")`. In Arm Compiler 6, you must also use the relevant `-ffixed-rN` `armclang` option. Alternatively, you can use equivalent inline assembler instructions.
`__inline(x)`	`__inline__`. The use of this keyword depends on the language mode.
`__int64`	You can use `int64_t`, which is a 64-bit integer type defined in the header file `<stdint.h>` (for C source files) or `<cstdint>` (for C++ source files). You can also use `long long`, however, if you use `long long` in C90 mode, the compiler gives: • a warning. • an error, if you also use `-pedantic-errors`.

Table 4-1 Keyword language extensions in Arm Compiler 5 and Arm Compiler 6 (continued)

Keyword supported by Arm Compiler 5	Recommended Arm Compiler 6 keyword or alternative
`__INTADDR__`	No equivalent.
`__irq`	`__attribute__((interrupt))`. This keyword is not supported in AArch64.
`__packed` for removing padding within structures.	`__attribute__((packed))`. This keyword provides limited functionality when compared to `__packed`: • The `__attribute__((packed))` variable attribute applies to members of a structure or union. It does not apply to variables that are not members of a structure or union. • `__attribute__((packed))` is not a type qualifier. Taking the address of a packed member can result in unaligned pointers, and usually the compiler generates a warning. Arm recommends upgrading this warning to an error when migrating code that uses `__packed`. To upgrade the warning to error, use the `armclang` option `-Werror=`*name*. The placement of the attribute is different from the placement of `__packed`. If your legacy code contains `typedef __packed struct`, then replace it with: `typedef struct __attribute__((packed))`
`__packed` as a type qualifier for unaligned access.	`__unaligned`. This keyword provides limited functionality when compared to the `__packed` type qualifier. The `__unaligned` type qualifier can be used over a structure only when using `typedef` or when declaring a structure variable. This limitation does not apply when using `__packed` in Arm Compiler 5. Therefore, there is no migration for legacy code that contains `__packed struct S{...};`.
`__pure`	`__attribute__((const))`
`__smc`	Use inline assembler instructions or equivalent routine.
`__softfp`	`__attribute__((pcs("aapcs")))`
`__svc`	Use inline assembler instructions or equivalent routine.
`__svc_indirect`	Use inline assembler instructions or equivalent routine.
`__svc_indirect_r7`	Use inline assembler instructions or equivalent routine.
`__thread`	`__thread` [COMMUNITY] You can use `__thread` only in conjunction with the `-femulated-tls` command-line option. When using this option, you must also provide an implementation of the function `__emutls_get_address`. For an example implementation, see *https://github.com/llvm/llvm-project/blob/master/compiler-rt/lib/builtins/emutls.c*
`__value_in_regs`	`__attribute__((value_in_regs))`
`__weak`	`__attribute__((weak))`
`__writeonly`	No equivalent.

Table 4-1 Keyword language extensions in Arm Compiler 5 and Arm Compiler 6 (continued)

Keyword supported by Arm Compiler 5	Recommended Arm Compiler 6 keyword or alternative
Named register variables for direct manipulation of a core register as if it were a C variable. For example: `register int R5 __asm("r5")`.	Use the `register` and `__asm` keywords for global named register variables using core registers. For example: `register int Reg5 __asm("r5")`. In Arm Compiler 6, you must also use the relevant `-ffixed-rN` `armclang` option.
Named register variables for direct manipulation of a system register, other than core registers, as if it were a C variable. For example: `register int fpscr __asm("fpscr")`.	No equivalent. To access `FPSCR`, use the `__vfp_status` intrinsic or inline assembly instructions.

Migrating the __packed keyword from Arm® Compiler 5 to Arm® Compiler 6

The `__packed` keyword in Arm Compiler 5 has the effect of:

- Removing the padding within structures.
- Qualifying the variable for unaligned access.

Arm Compiler 6 does not support `__packed`, but supports `__attribute__((packed))` and `__unaligned` keyword. Depending on the use, you might need to replace `__packed` with both `__attribute__((packed))` and `__unaligned`. The following table shows the migration paths for various uses of `__packed`.

Table 4-2 Migrating the __packed keyword

Arm Compiler 5	Arm Compiler 6
`__packed int x;`	`__unaligned int x;`
`__packed int *x;`	`__unaligned int *x;`
`int * __packed x;`	`int * __unaligned x;`
`__unaligned int * __packed x;`	`__unaligned int * __unaligned x;`
`typedef __packed struct S{...} s;`	`typedef __unaligned struct __attribute__((packed)) S{...} s;`
`__packed struct S{...};`	There is no migration. Use a `typedef` instead.
`__packed struct S{...} s;`	`__unaligned struct __attribute__((packed)) S{...} s;` Subsequent declarations of variables of type `struct S` must use `__unaligned`, for example `__unaligned struct S s2`.
`struct S{__packed int a;}`	`struct S {__attribute__((packed)) __unaligned int a;}`

Related references

4.6 C and C++ implementation compatibility on page 4-70

4.2 Language extension compatibility: attributes on page 4-58

4.3 Language extension compatibility: pragmas on page 4-61

Related information

-W

4.2 Language extension compatibility: attributes

Arm Compiler 6 supports some function, variable, and type attributes that were supported in Arm Compiler 5. Other attributes are not supported, or have an alternate implementation.

Arm Compiler 5 and Arm Compiler 6 support the following attributes. These attributes do not require modification in your code:

Note

The `__declspec` keyword is deprecated.

- `__attribute__((aligned(x)))`
- `__attribute__((const))`
- `__attribute__((deprecated))`
- `__attribute__((noinline))`
- `__declspec(noinline)`
- `__attribute__((nonnull))`
- `__attribute__((noreturn))`
- `__declspec(noreturn)`
- `__attribute__((nothrow))`
- `__declspec(nothrow)`
- `__attribute__((pcs("calling convention")))`
- `__attribute__((pure))`
- `__attribute__((unused))`
- `__attribute__((used))`
- `__attribute__((visibility))`
- `__attribute__((weak))`
- `__attribute__((weakref))`

The following Arm Compiler 5 attributes are not supported by Arm Compiler 6:

- `__attribute__((nomerge))`
- `__attribute__((notailcall))`

However, since Arm Compiler 6 is built on LLVM technology and preserves the functionality of that technology where possible, you may consider using the following community (open-source Clang) features instead:

- `__attribute__((nomerge))`
- `__attribute__((not_tail_called))`

Community features on page 1-14 are not supported by Arm and are used at your own risk. You are responsible for making sure that any generated code using community features is operating correctly.

Though Arm Compiler 6 supports certain `__declspec` attributes, Arm recommends using `__attribute__` where available.

Table 4-3 Support for __declspec attributes

declspec supported by Arm Compiler 5	Recommended Arm Compiler 6 alternative
`__declspec(dllimport)`	None. There is no support for BPABI linking models.
`__declspec(dllexport)`	None. There is no support for BPABI linking models.
`__declspec(noinline)`	`__attribute__((noinline))`
`__declspec(noreturn)`	`__attribute__((noreturn))`
`__declspec(nothrow)`	`__attribute__((nothrow))`

Table 4-3 Support for __declspec attributes (continued)

declspec supported by Arm Compiler 5	Recommended Arm Compiler 6 alternative
`__declspec(notshared)`	None. There is no support for BPABI linking models.
`__declspec(thread)`	`__thread`

__attribute__((always_inline))

Arm Compiler 5 and Arm Compiler 6 support `__attribute__((always_inline))`. However, this attribute might require modification in your code.

When using Arm Compiler 5, `__attribute__((always_inline))` affects the linkage of the function according to the inline semantics of the source language.

When using Arm Compiler 6, `__attribute__((always_inline))` does not affect the linkage of the function. To change the linkage according to the inline semantics, you must use the keyword `inline` or `__inline__` (for C90). For more information, see *__attribute__((always_inline))*.

__attribute__((section("*name*")))

Arm Compiler 5 and Arm Compiler 6 support `__attribute__((section("name")))`. However, this attribute might require modification in your code.

When using Arm Compiler 5, section names do not need to be unique. Therefore, you could use the same section name to create different section types.

Arm Compiler 6 supports multiple sections with the same section name only if you specify a *unique ID*. You must ensure that different section types either:

- Have a unique section name.
- Have a unique ID, if they have the same section name.

If you use the same section name, for another section or symbol, without a unique ID, then `armclang` integrated assembler merges the sections and gives the merged section the flags of the first section with that name.

Migrating __attribute__((at(*address*))) and zero-initialized __attribute__((section("*name*"))) from Arm® Compiler 5 to Arm® Compiler 6

Arm Compiler 5 supports the following attributes, which Arm Compiler 6 does not support:

- `__attribute__((at(address)))` to specify the absolute address of a function or variable.
- `__attribute__((at(address), zero_init))` to specify the absolute address of a zero-initialized variable.
- `__attribute__((section(name), zero_init))` to place a zero-initialized variable in a zero-initialized section with the given *name*.
- `__attribute__((zero_init))` to generate an error if the variable has an initializer.

The following table shows migration paths for these features using Arm Compiler 6 supported features:

Table 4-4 Migrating __attribute__((at(*address*))) and zero-initialized __attribute__((section("*name*")))

Arm Compiler 5 attribute	Arm Compiler 6 attribute	Description
`__attribute__((at(`*`address`*`)))`	`__attribute__((section(".ARM.__at_`*`address`*`")))`	`armlink` in Arm Compiler 6 still supports the placement of sections in the form of `.ARM.__at_`*`address`* **Note** The Arm Compiler 6 attribute only supports a string to specify the section. To use an arithmetic expression, see *Supporting arithmetic expressions in the at(address) attribute in Arm® Compiler 6* on page 4-60.
`__attribute__((at(`*`address`*`), zero_init))`	`__attribute__((section(".bss.ARM.__at_`*`address`*`")))`	`armlink` in Arm Compiler 6 supports the placement of zero-initialized sections in the form of `.bss.ARM.__at_`*`address`*. The `.bss` prefix is case-sensitive and must be all lowercase.
`__attribute__((section(`*`name`*`), zero_init))`	`__attribute__((section(".bss.`*`name`*`")))`	*`name`* is a name of your choice. The `.bss` prefix is case-sensitive and must be all lowercase.
`__attribute__((zero_init))`	Arm Compiler 6 by default places zero-initialized variables in a `.bss` section. However, there is no equivalent to generate an error when you specify an initializer.	If the variable has an initializer, Arm Compiler 5 generates an error. Otherwise, it places the zero-initialized variable in a `.bss` section.

Supporting arithmetic expressions in the at(address) attribute in Arm® Compiler 6

The `at(address)` attribute in Arm Compiler 5 supports arithmetic expressions to specify the section, for example:

```
my_variable_type my_variable __attribute__((at(0xE0001000 + MY_PREDEFINED_OFFSET)));
```

To do the equivalent in Arm Compiler 6, you must use a pointer approach. For this Arm Compiler 5 example, change the code to:

```
static my_variable_type * const my_address = (my_variable_type *) (0xE0001000 +
MY_PREDEFINED_OFFSET);

#define my_variable (*my_address)
```

Related references

4.6 C and C++ implementation compatibility on page 4-70

4.1 Language extension compatibility: keywords on page 4-55

4.3 Language extension compatibility: pragmas on page 4-61

Related information

Placing __at sections at a specific address

4.3 Language extension compatibility: pragmas

Arm Compiler 6 provides support for some pragmas that are supported in Arm Compiler 5. Other pragmas are not supported, or must be replaced with alternatives.

The following table lists some of the commonly used pragmas that are supported by Arm Compiler 5 but are not supported by Arm Compiler 6. Replace any instances of these pragmas in your code with the recommended alternative.

Table 4-5 Pragma language extensions that must be replaced

<table>
<tr><th>Pragma supported by Arm Compiler 5</th><th>Recommended Arm Compiler 6 alternative</th></tr>
<tr><td><code>#pragma import (<i>symbol</i>)</code></td><td><pre>__asm(".global <i>symbol</i>\n\t");</pre></td></tr>
<tr><td><code>#pragma anon_unions</code>
<code>#pragma no_anon_unions</code></td><td>In C, anonymous structs and unions are a C11 extension which is enabled by default in armclang. If you specify the <code>-pedantic</code> option, the compiler emits warnings about extensions do not match the specified language standard. For example:
<pre>armclang --target=aarch64-arm-none-eabi -c -pedantic --std=c90
test.c
  test.c:3:5: warning: anonymous structs are a C11 extension [-
Wc11-extensions]</pre>
In C++, anonymous unions are part of the language standard, and are always enabled. However, anonymous structs and classes are an extension. If you specify the <code>-pedantic</code> option, the compiler emits warnings about anonymous structs and classes. For example:
<pre>armclang --target=aarch64-arm-none-eabi -c -pedantic -xc++
test.c
  test.c:3:5: warning: anonymous structs are a GNU extension [-
Wgnu-anonymous-struct]</pre>
Introducing anonymous unions, struct and classes using a <code>typedef</code> is a separate extension in <code>armclang</code>, which must be enabled using the <code>-fms-extensions</code> option.</td></tr>
<tr><td><code>#pragma arm</code>
<code>#pragma thumb</code></td><td><code>armclang</code> does not support switching instruction set in the middle of a file. You can use the command-line options <code>-marm</code> and <code>-mthumb</code> to specify the instruction set of the whole file.</td></tr>
<tr><td><code>#pragma arm section</code></td><td><code>#pragma clang section</code>
In Arm Compiler 5, the section types you can use this pragma with are <code>rodata</code>, <code>rwdata</code>, <code>zidata</code>, and <code>code</code>. In Arm Compiler 6, the equivalent section types are <code>rodata</code>, <code>data</code>, <code>bss</code>, and <code>text</code> respectively.</td></tr>
</table>

Table 4-5 Pragma language extensions that must be replaced (continued)

Pragma supported by Arm Compiler 5	Recommended Arm Compiler 6 alternative

`#pragma diag_default`
`#pragma diag_suppress`
`#pragma diag_remark`
`#pragma diag_warning`
`#pragma diag_error`
	The following pragmas provide equivalent functionality for `diag_suppress`, `diag_warning`, and `diag_error`:
	• `#pragma clang diagnostic ignored "-Wmultichar"`
	• `#pragma clang diagnostic warning "-Wmultichar"`
	• `#pragma clang diagnostic error "-Wmultichar"`
	Note that these pragmas use `armclang` diagnostic groups, which do not have a precise mapping to `armcc` diagnostic tags.
	`armclang` has no equivalent to `diag_default` or `diag_remark`. `diag_default` can be replaced by wrapping the change of diagnostic level with `#pragma clang diagnostic push` and `#pragma clang diagnostic pop`, or by manually returning the diagnostic to the default level.
	There is an additional diagnostic level supported in armclang, `fatal`, which causes compilation to fail without processing the rest of the file. You can set this as follows:
	`#pragma clang diagnostic fatal "-Wmultichar"`

`#pragma exceptions_unwind` `#pragma no_exceptions_unwind`	`armclang` does not support these pragmas. Use the `__attribute__((nothrow))` function attribute instead.
`#pragma GCC system_header`	This pragma is supported by both `armcc` and `armclang`, but `#pragma clang system_header` is the preferred spelling in `armclang` for new code.
`#pragma hdrstop` `#pragma no_pch`	`armclang` does not support these pragmas.
`#pragma import(__use_no_semihosting)` `#pragma import(__use_no_semihosting_swi)`	`armclang` does not support these pragmas. However, in C code, you can replace these pragmas with: `__asm(".global __use_no_semihosting\n\t");`
`#pragma inline` `#pragma no_inline`	`armclang` does not support these pragmas. However, inlining can be disabled on a per-function basis using the `__attribute__((noinline))` function attribute. The default behavior of both `armcc` and `armclang` is to inline functions when the compiler considers this worthwhile, and this is the behavior selected by using `#pragma inline` in `armcc`. To force a function to be inlined in `armclang`, use the `__attribute__((always_inline))` function attribute.

`#pragma Onum`
`#pragma Ospace`
`#pragma Otime`
	`armclang` does not support changing optimization options within a file. Instead these must be set on a per-file basis using command-line options.

Table 4-5 Pragma language extensions that must be replaced (continued)

Pragma supported by Arm Compiler 5	Recommended Arm Compiler 6 alternative
`#pragma pop` `#pragma push`	`armclang` does not support these pragmas. Therefore, you cannot push and pop the state of all supported pragmas. However, you can push and pop the state of the diagnostic pragmas and the state of the pack pragma. To control the state of the diagnostic pragmas, use `#pragma clang diagnostic push` and `#pragma clang diagnostic pop`. To control the state of the pack pragma, use `#pragma pack(push)` and `#pragma pack(pop)`.
`#pragma softfp_linkage`	`armclang` does not support this pragma. Instead, use the `__attribute__((pcs("aapcs")))` function attribute to set the calling convention on a per-function basis, or use the `-mfloat-abi=soft` command-line option to set the calling convention on a per-file basis.
`#pragma no_softfp_linkage`	`armclang` does not support this pragma. Instead, use the `__attribute__((pcs("aapcs-vfp")))` function attribute to set the calling convention on a per-function basis, or use the `-mfloat-abi=hard` command-line option to set the calling convention on a per-file basis.
`#pragma unroll[(`*n*`)]` `#pragma unroll_completely`	`armclang` supports these pragmas. The default for `#pragma unroll` (that is, with no iteration count specified) differs between `armclang` and `armcc`: • With `armclang`, the default is to fully unroll a loop. • With `armcc`, the default is `#pragma unroll(4)`.

Related references

4.6 C and C++ implementation compatibility on page 4-70
4.1 Language extension compatibility: keywords on page 4-55
4.2 Language extension compatibility: attributes on page 4-58
4.5 Diagnostics for pragma compatibility on page 4-68

Related information

Reference Guide: #pragma GCC system_header
Reference Guide: #pragma once
Reference Guide: #pragma pack(n)
Reference Guide: #pragma weak symbol, #pragma weak symbol1 = symbol2
Reference Guide: #pragma unroll[(n)], #pragma unroll_completely

4.4 Language extension compatibility: intrinsics

Arm Compiler 6 provides support for some intrinsics that are supported in Arm Compiler 5.

The following table lists some of the commonly used intrinsics that are supported by Arm Compiler 5 and shows whether Arm Compiler 6 supports them or provides an alternative. If there is no support in Arm Compiler 6, you must replace them with suitable inline assembly instructions or calls to the standard library. To use the intrinsic in Arm Compiler 6, you must include the appropriate header file. The ACLE intrinsics that are supported by Arm Compiler 5 are described in the *Arm® C Language Extensions 2.1*. For more information on the ACLE intrinsics that are supported by Arm Compiler 6, see the latest *Arm® C Language Extensions*.

Note

- This is not an exhaustive list of all the intrinsics.
- The intrinsics provided in `<arm_compat.h>` are only supported for AArch32.

Table 4-6 Compiler intrinsic support in Arm Compiler 6

Intrinsic in Arm Compiler 5	Function	Support in Arm Compiler 6	Header file for Arm Compiler 6
`__breakpoint`	Inserts a `BKPT` instruction.	Yes	`arm_compat.h`
`__cdp`	Inserts a coprocessor instruction.	Yes. In Arm Compiler 6, the equivalent intrinsic is `__arm_cdp`.	`arm_acle.h`
`__clrex`	Inserts a `CLREX` instruction.	No	-
`__clz`	Inserts a `CLZ` instruction or equivalent routine.	Yes	`arm_acle.h`
`__current_pc`	Returns the program counter at this point.	Yes	`arm_compat.h`
`__current_sp`	Returns the stack pointer at this point.	Yes	`arm_compat.h`
`__isb`	Inserts `ISB` or equivalent.	Yes	`arm_acle.h`
`__disable_fiq`	Disables FIQ interrupts (Armv7 architecture only). Returns previous value of FIQ mask.	Yes	`arm_compat.h`
`__disable_irq`	Disable IRQ interrupts. Returns previous value of IRQ mask.	Yes	`arm_compat.h`
`__dmb`	Inserts a `DMB` instruction or equivalent.	Yes	`arm_acle.h`
`__dsb`	Inserts a `DSB` instruction or equivalent.	Yes	`arm_acle.h`
`__enable_fiq`	Enables fast interrupts.	Yes	`arm_compat.h`
`__enable_irq`	Enables IRQ interrupts.	Yes	`arm_compat.h`
`__fabs`	Inserts a `VABS` or equivalent code sequence.	No. Arm recommends using the standard C library function `fabs()`.	-
`__fabsf`	Single precision version of `__fabs`.	No. Arm recommends using the standard C library function `fabsf()`.	-

Table 4-6 Compiler intrinsic support in Arm Compiler 6 (continued)

Intrinsic in Arm Compiler 5	Function	Support in Arm Compiler 6	Header file for Arm Compiler 6
`__force_stores`	Flushes all external variables visible from this function, if they have been changed.	Yes	`arm_compat.h`
`__ldrex`	Inserts an appropriately sized Load Exclusive instruction.	No. This intrinsic is deprecated in ACLE 2.0.	-
`__ldrexd`	Inserts an `LDREXD` instruction.	No. This intrinsic is deprecated in ACLE 2.0.	-
`__ldrt`	Inserts an appropriately sized user-mode load instruction.	No	-
`__memory_changed`	Is similar to `__force_stores`, but also reloads the values from memory.	Yes	`arm_compat.h`
`__nop`	Inserts a `NOP` or equivalent instruction that will not be optimized away. It also inserts a sequence point, and scheduling barrier for side-effecting function calls.	Yes	`arm_acle.h`
`__pld`	Inserts a PLD instruction, if supported.	Yes	`arm_acle.h`
`__pldw`	Inserts a `PLDW` instruction, if supported (Armv7 architecture with MP).	No. Arm recommends using `__pldx` described in the ACLE document.	`arm_acle.h`
`__pli`	Inserts a PLI instruction, if supported.	Yes	`arm_acle.h`
`__promise`	Compiler assertion that the expression always has a nonzero value. If asserts are enabled then the promise is checked at runtime by evaluating *`expr`* using `assert(`*`expr`*`)`.	Yes. However, you must `#include <assert.h>` to use `__promise`. `__promise` has the same behavior as `assert()` unless at least one of `NDEBUG` or `__DO_NOT_LINK_PROMISE_WITH_ASSERT` is defined.	`assert.h`
`__qadd`	Inserts a saturating add instruction, if supported.	Yes	`arm_acle.h`
`__qdbl`	Inserts instructions equivalent to `qadd(val,val)`, if supported.	Yes	`arm_acle.h`
`__qsub`	Inserts a saturating subtract, or equivalent routine, if supported.	Yes	`arm_acle.h`
`__rbit`	Inserts a bit reverse instruction.	Yes	`arm_acle.h`
`__rev`	Insert a `REV`, or endian swap instruction.	Yes	`arm_acle.h`
`__return_address`	Returns value of `LR` when returning from current function, without inhibiting optimizations like inlining or tailcalling.	No. Arm recommends using inline assembly instructions.	-

Table 4-6 Compiler intrinsic support in Arm Compiler 6 (continued)

Intrinsic in Arm Compiler 5	Function	Support in Arm Compiler 6	Header file for Arm Compiler 6
`__ror`	Insert an `ROR` instruction.	Yes	`arm_acle.h`
`__schedule_barrier`	Create a sequence point without effecting memory or inserting `NOP` instructions. Functions with side effects cannot move past the new sequence point.	Yes	`arm_compat.h`
`__semihost`	Inserts an `SVC` or `BKPT` instruction.	Yes	`arm_compat.h`
`__sev`	Insert a `SEV` instruction. Error if the `SEV` instruction is not supported.	Yes	`arm_acle.h`
`__sqrt`	Inserts a `VSQRT` instruction on targets with a VFP coprocessor.	No	-
`__sqrtf`	single precision version of `__sqrt`.	No	-
`__ssat`	Inserts an `SSAT` instruction. Error if the `SSAT` instruction is not supported.	Yes	`arm_acle.h`
`__strex`	Inserts an appropriately sized Store Exclusive instruction.	No. This intrinsic is deprecated in ACLE 2.0.	-
`__strexd`	Inserts a doubleword Store Exclusive instruction.	No. This intrinsic is deprecated in ACLE 2.0.	-
`__strt`	Insert an appropriately sized `STRT` instruction.	No	-
`__swp`	Inserts an appropriately sized `SWP` instruction.	Yes. However, the `SWP` instruction is deprecated, and Arm does not recommend the use of `__swp`.	`arm_acle.h`
`__usat`	Inserts a `USAT` instruction. Error if the `USAT` instruction is not supported.	Yes	`arm_acle.h`
`__wfe`	Inserts a `WFE` instruction. Error if the `WFE` instruction is not supported.	Yes	`arm_acle.h`
`__wfi`	Inserts a `WFI` instruction. Error if the `WFI` instruction is not supported.	Yes	`arm_acle.h`
`__yield`	Inserts a `YIELD` instruction. Error if the `YIELD` instruction is not supported.	Yes	`arm_acle.h`
`ARMv6 SIMD intrinsics`	Inserts an Armv6 SIMD instruction.	No	-

Table 4-6 Compiler intrinsic support in Arm Compiler 6 (continued)

Intrinsic in Arm Compiler 5	Function	Support in Arm Compiler 6	Header file for Arm Compiler 6
`ETSI intrinsics`	35 intrinsic functions and 2 global variable flags specified in ETSI G729 used for speech encoding. These are provided in the Arm headers in `dspfns.h`.	No	-
`C55x intrinsics`	Emulation of selected TI C55x compiler intrinsics.	No	-
`__vfp_status`	Reads the FPSCR.	Yes	`arm_compat.h`
`FMA intrinsics`	Intrinsics for fused-multiply-add on the Cortex-M4 or Cortex-A5 processor in c99 mode.	No	-

4.5 Diagnostics for pragma compatibility

Older `armcc` compiler versions supported many pragmas which are not supported by `armclang`, but which could change the semantics of code. When `armclang` encounters these pragmas, it generates diagnostic messages.

The following table shows which diagnostics are generated for each pragma type, and the diagnostic group to which that diagnostic belongs. `armclang` generates diagnostics as follows:

- Errors indicate use of an `armcc` pragma which could change the semantics of code.
- Warnings indicate use of any other `armcc` pragma which is ignored by armclang.
- Pragmas other than those listed are silently ignored.

Table 4-7 Pragma diagnostics

Pragma supported by older compiler versions	Default diagnostic type	Diagnostic group
`#pragma anon_unions`	Warning	`armcc-pragma-anon-unions`
`#pragma no_anon_unions`	Warning	`armcc-pragma-anon-unions`
`#pragma arm`	Error	`armcc-pragma-arm`
`#pragma arm section [`*`section_type_list`*`]`	Error	`armcc-pragma-arm`
`#pragma diag_default `*`tag`*`[,`*`tag`*`,...]`	Error	`armcc-pragma-diag`
`#pragma diag_error `*`tag`*`[,`*`tag`*`,...]`	Error	`armcc-pragma-diag`
`#pragma diag_remark `*`tag`*`[,`*`tag`*`,...]`	Warning	`armcc-pragma-diag`
`#pragma diag_suppress `*`tag`*`[,`*`tag`*`,...]`	Warning	`armcc-pragma-diag`
`#pragma diag_warning `*`tag`*`[,`*`tag`*`,...]`	Warning	`armcc-pragma-diag`
`#pragma exceptions_unwind`	Error	`armcc-pragma-exceptions-unwind`
`#pragma no_exceptions_unwind`	Error	`armcc-pragma-exceptions-unwind`
`#pragma GCC system_header`	None	-
`#pragma hdrstop`	Warning	`armcc-pragma-hdrstop`
`#pragma import `*`symbol_name`*	Error	`armcc-pragma-import`
`#pragma inline`	Warning	`armcc-pragma-inline`
`#pragma no_inline`	Warning	`armcc-pragma-inline`
`#pragma no_pch`	Warning	`armcc-pragma-no-pch`
`#pragma O`*`num`*	Warning	`armcc-pragma-optimization`
`#pragma once`	None	-
`#pragma Ospace`	Warning	`armcc-pragma-optimization`
`#pragma Otime`	Warning	`armcc-pragma-optimization`
`#pragma pack`	None	-
`#pragma pop`	Error	`armcc-pragma-push-pop`
`#pragma push`	Error	`armcc-pragma-push-pop`
`#pragma softfp_linkage`	Error	`armcc-pragma-softfp-linkage`
`#pragma no_softfp_linkage`	Error	`armcc-pragma-softfp-linkage`
`#pragma thumb`	Error	`armcc-pragma-thumb`

Table 4-7 Pragma diagnostics (continued)

Pragma supported by older compiler versions	Default diagnostic type	Diagnostic group
`#pragma weak` *`symbol`*	None	-
`#pragma weak` *`symbol1`* `=` *`symbol2`*	None	-

In addition to the above diagnostic groups, there are the following additional diagnostic groups:

`armcc-pragmas`
Contains all of the above diagnostic groups.

`unknown-pragmas`
Contains diagnostics about pragmas which are not known to armclang, and are not in the above table.

`pragmas`
Contains all pragma-related diagnostics, including `armcc-pragmas` and `unknown-pragmas`.

Any non-fatal `armclang` diagnostic group can be ignored, upgraded, or downgraded using the following command-line options:

Suppress a group of diagnostics:
`-Wno-`*`diag-group`*

Upgrade a group of diagnostics to warnings:
`-Wdiag-group`

Upgrade a group of diagnostics to errors:
`-Werror=`*`diag-group`*

Downgrade a group of diagnostics to warnings:
`-Wno-error=`*`diag-group`*

Related references

4.3 Language extension compatibility: pragmas on page 4-61

4.6 C and C++ implementation compatibility

Arm Compiler 6 C and C++ implementation details differ from previous compiler versions.

The following table describes the C and C++ implementation detail differences.

Table 4-8 C and C++ implementation detail differences

Feature	Older versions of Arm Compiler	Arm Compiler 6
Integer operations		

Shifts	`int` *`shifts`* `> 0 && < 127`
	`int` *`left_shifts`* `> 31 == 0`
	`int` *`right_shifts`* `> 31 == 0`
	(for unsigned or positive)
	`int` *`right_shifts`* `> 31 == -1`
	(for negative)
	`long long` *`shifts`* `> 0 && < 63`
		Warns when shift amount > width of type.
		You can use the `-Wshift-count-overflow` option to suppress this warning.

Integer division	Checks that the sign of the remainder matches the sign of the numerator	The sign of the remainder is not necessarily the same as the sign of the numerator.
Floating-point operations		
Default standard	IEEE 754 standard, rounding to nearest representable value, exceptions disabled by default.	All facilities, operations, and representations guaranteed by the IEEE standard are available in single and double-precision. Modes of operation can be selected dynamically at runtime. This is equivalent to the `--fpmode=ieee_full` option in older versions of Arm Compiler.
`#pragma STDC FP_CONTRACT`	`#pragma STDC FP_CONTRACT`	Might affect code generation.
Unions, enums and structs		
Enum packing	Enums are implemented in the smallest integral type of the correct sign to hold the range of the enum values, except for when compiling in C++ mode with `--enum_is_int`.	By default enums are implemented as **`int`**, with **`long long`** used when required.
Allocation of bit-fields in containers	Allocation of bit-fields in containers.	A container is an object, aligned as the declared type. Its size is sufficient to contain the bit-field, but might be smaller or larger than the bit-field declared type.
Signedness of plain bit-fields	Unsigned. Plain bit-fields declared without either the **`signed`** or **`unsigned`** qualifiers default to **`unsigned`**. The `--signed_bitfields` option treats plain bit-fields as **`signed`**.	Signed. Plain bit-fields declared without either the **`signed`** or **`unsigned`** qualifiers default to **`signed`**. There is no equivalent to either the `--signed_bitfields` or `--no_signed_bitfields` options.
Arrays and pointers		

Table 4-8 C and C++ implementation detail differences (continued)

Feature	Older versions of Arm Compiler	Arm Compiler 6
Casting between integers and pointers	No change of representation	Converting a signed integer to a pointer type with greater bit width sign-extends the integer. Converting an unsigned integer to a pointer type with greater bit width zero-extends the integer.
Misc C		
`sizeof(wchar_t)`	2 bytes	4 bytes
`size_t`	Defined as **`unsigned int`**, 32-bit.	Defined as **`unsigned int`** in 32-bit architectures, and <*`sign`*><*`type`*> 64-bit in 64-bit architectures.
`ptrdiff_t`	Defined as **`signed int`**, 32-bit.	Defined as **`unsigned int`** in 32-bit architectures, and <*`sign`*><*`type`*> 64-bit in 64-bit architectures.
Misc C++		
C++ library	Rogue Wave Standard C++ Library	LLVM libc++ Library **Note** • When the C++ library is used in source code, there is limited compatibility between object code created with Arm Compiler 6 and object code created with Arm Compiler 5. This also applies to indirect use of the C++ library, for example memory allocation or exception handling.
Implicit inclusion	If compilation requires a template definition from a template declared in a header file `xyz.h`, the compiler implicitly includes the file `xyz.cc` or `xyz.CC`.	Not supported.
Alternative template lookup algorithms	When performing referencing context lookups, name lookup matches against names from the instantiation context as well as from the template definition context.	Not supported.
Exceptions	Off by default, function unwinding on with `--exceptions` by default.	On by default in C++ mode. **Note** For C++ code, `-fexceptions` has a large increase in the code size. If you use `-fno_exceptions`, then the code size is in the range of that created with Arm Compiler 5.
Translation		
Diagnostics messages format	`source-file, line-number : severity : error-code : explanation`	`source-file:line-number:char-number: description [diagnostic-flag]`
Environment		
Physical source file bytes interpretation	Current system locale dependent or set using the `--locale` command-line option.	UTF-8, either with or without the *Byte Order Mark* (BOM).

Related references

4.1 Language extension compatibility: keywords on page 4-55

4.2 Language extension compatibility: attributes on page 4-58

4.3 Language extension compatibility: pragmas on page 4-61

4.7 Compatibility of C++ objects on page 4-73

Related information

Locale support in Arm Compiler

4.7 Compatibility of C++ objects

The compatibility of C++ objects compiled with Arm Compiler 5 depends on the C++ libraries used.

Compatibility with objects compiled using Rogue Wave standard library headers

Arm Compiler 6 does not support binary compatibility with objects compiled using the Rogue Wave standard library include files.

There are warnings at link time when objects are mixed. `L6869W` is reported if an object requests the Rogue Wave standard library. `L6870W` is reported when using an object that is compiled with Arm Compiler 5 with exceptions support.

The impact of mixing objects that have been compiled against different C++ standard library headers might include:
- Undefined symbol errors.
- Increased code size.
- Possible runtime errors.

If you have Arm Compiler 6 objects that have been compiled with the legacy `-stdlib=legacy_cpplib` option then these objects use the Rogue Wave standard library and therefore might be incompatible with objects created using Arm Compiler 6.4 or later. To resolve these issues, you must recompile all object files with Arm Compiler 6.4 or later.

Compatibility with C++ objects compiled using Arm® Compiler 5

The choice of C++ libraries at link time must match the choice of C++ include files at compile time for all input objects. Arm Compiler 5 objects that use the Rogue Wave C++ libraries are not compatible with Arm Compiler 6 objects. Arm Compiler 5 objects that use C++ but do not make use of the Rogue Wave header files can be compatible with Arm Compiler 6 objects that use libc++ but this is not guaranteed.

Arm recommends using Arm Compiler 6 for building the object files.

Compatibility of arrays of objects compiled using Arm® Compiler 5

Arm Compiler 6 is not compatible with objects from Arm Compiler 5 that use operator `new[]` and `delete[]`. Undefined symbol errors result at link time because Arm Compiler 6 does not provide the helper functions that Arm Compiler 5 depends on. For example:

```
construct.cpp:

class Foo
{
public:
    Foo() : x_(new int) { *x_ = 0; }
    void setX(int x) { *x_ = x; }
    ~Foo() { delete x_; }
private:
    int* x_;
};

void func(void)
{
    Foo* array;
    array = new Foo [10];
    array[0].setX(1);
    delete[] array;
}
```

If you build this example with the Arm Compiler 5 compiler, `armcc`, and link with the Arm Compiler 6 linker, `armlink`, using:

```
armcc -c construct.cpp -Ospace -O1 --cpu=cortex-a9
armlink construct.o -o construct.axf
```

the linker reports:

```
Error: L6218E: Undefined symbol __aeabi_vec_delete (referred from construct.o).
Error: L6218E: Undefined symbol __aeabi_vec_new_cookie_nodtor (referred from construct.o).
```

To resolve these linker errors, you must use the Arm Compiler 6 compiler, `armclang`, to compile all C++ files that use the `new[]` and `delete[]` operators.

Note

You do not have to specify `--stdlib=libc++` for `armlink`, because this is the default and only option in Arm Compiler 6.4, and later.

Related information

--stdlib

Chapter 5

Migrating from armasm to the armclang Integrated Assembler

Describes how to migrate assembly code from `armasm` syntax to GNU syntax (used by `armclang`).

It contains the following sections:

5.1 Migration of assembler command-line options from armasm to the armclang integrated assembler

Arm Compiler 6 provides many command-line options, including most Clang command-line options as well as several Arm-specific options.

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page 1-14.

Note

The following GNU assembly directives are [COMMUNITY] features:
- `.eabi_attribute Tag_ABI_PCS_RO_data, value`
- `.eabi_attribute Tag_ABI_PCS_R9_use, value`
- `.eabi_attribute Tag_ABI_PCS_RW_data, value`
- `.eabi_attribute Tag_ABI_VFP_args, value`
- `.eabi_attribute Tag_CPU_unaligned_access, value`
- `.ident`
- `.protected`
- `.section .note.GNU-stack, "x"`
- `-Wa,--noexecstack`
- `-Wa,-L`
- `-Wa,-defsym,symbol=value`

The following table describes the most common `armasm` command-line options, and shows the equivalent options for the `armclang` integrated assembler.

Additional information about command-line options is available:
- The *Arm® Compiler Reference Guide* provides more detail about the command-line options.
- For a full list of Clang command-line options, consult the Clang and LLVM documentation.

Table 5-1 Comparison of command-line options in `armasm` and the `armclang` integrated assembler

`armasm` option	`armclang` integrated assembler option	Description
`--arm_only`	No equivalent.	Enforces A32 instructions only.
`--apcs=/nointerwork`	No equivalent.	Specifies that the code in the input file can interwork between A32 and T32 safely. Interworking is always enabled in Arm Compiler 6.
`--apcs=/ropi`, `--apcs=/noropi`	No direct equivalent.	With `armasm`, the options specify whether the code in the input file is *Read-Only Position-Independent* (ROPI) code. With the `armclang` integrated assembler, use the GNU assembly `.eabi_attribute` directive instead. To specify that the code is ROPI code, use the directive as follows: `.eabi_attribute Tag_ABI_PCS_RO_data, 1` The code is marked as not ROPI code by default.

Table 5-1 Comparison of command-line options in `armasm` and the `armclang` integrated assembler (continued)

`armasm` **option**	`armclang` **integrated assembler option**	**Description**

`--apcs=/rwpi`,
`--apcs=/norwpi`
	No direct equivalent.	With `armasm`, the options specify whether the code in the input file is *Read-Write Position-Independent* (RWPI) code.
		With the `armclang` integrated assembler, use the GNU assembly `.eabi_attribute` directive instead.
		To specify that the code is RWPI code, use the directive as follows:
		`.eabi_attribute Tag_ABI_PCS_R9_use,  1`
		`.eabi_attribute Tag_ABI_PCS_RW_data, 2`
		The code is marked as not RWPI code by default.

`--apcs=/hardfp`,
`--apcs=/softfp`
	No direct equivalent.	With `armasm`, the options set attributes in the object file to request hardware or software floating-point linkage.
		With the `armclang` integrated assembler, use the GNU assembly `.eabi_attribute` directive instead.
		To request hardware floating-point linkage, use the directive as follows:
		`.eabi_attribute Tag_ABI_VFP_args, 1`
		To request software floating-point linkage, use the directive as follows:
		`.eabi_attribute Tag_ABI_VFP_args, 0`

`--checkreglist`, `--diag_warning=1206`	This is the default.	Generates warnings if register lists in `LDM` and `STM` instructions are not provided in increasing register number order. **Note** This warning cannot be suppressed or upgraded to an error.
`--comment_section`, `--no_comment_section`	No direct equivalent.	With `armasm`, the option controls the inclusion of a comment section `.comment` in object files. With the `armclang` integrated assembler, use the GNU assembly `.ident` directive to manually add a comment section.
`--debug,` `-g`	`-g`	Instructs the assembler to generate DWARF debug tables. With `armasm`, the default format for debug tables is DWARF 3. Named local labels are not preserved in the object file, unless the `--keep` option is used. With the `armclang` integrated assembler, the default format for debug tables is DWARF 4. Named local labels are always preserved in the object file. See the entry for `--keep` in this table for details.

Table 5-1 Comparison of command-line options in `armasm` and the `armclang` integrated assembler (continued)

`armasm` **option**	`armclang` **integrated assembler option**	**Description**
`--diag_warning=1645`	No equivalent.	With `armasm`, the option enables warnings about instruction substitutions. With the `armclang` integrated assembler, instruction substitution support is limited. Where it is not supported, the assembler generates an error message. Use the `armasm` warning when migrating code to find instructions being substituted and perform the substitution manually.
`--diag_warning=1763`	No equivalent.	With `armasm`, the option enables warnings about automatic generation of `IT` blocks when assembling T32 code (formerly Thumb code). With the `armclang` integrated assembler, automatic generation of `IT` blocks is disabled by default. The assembler generates an error message when assembling conditional instructions without an enclosing `IT` block. To enable automatic generation of `IT` blocks, use the command-line option `-mimplicit-it=always` or `-mimplicit-it=thumb`.
`--dllexport_all`	No direct equivalent.	With `armasm`, the option gives all exported global symbols `STV_PROTECTED` visibility in `ELF` rather than `STV_HIDDEN`, unless overridden by source directives. With the `armclang` integrated assembler, use the GNU assembly `.protected` directive to manually give exported symbols `STV_PROTECTED` visibility.
`--execstack`, `--no_execstack`	`-Wa,--noexecstack` No direct equivalent for `--execstack`.	With `armasm`, the option generates a `.note.GNU-stack` section marking the stack as either executable or non-executable. With the `armclang` integrated assembler, the equivalent option can be used to generate a `.note.GNU-stack` section marking the stack as non-executable. To generate such a section and mark the stack as executable, use the GNU assembly `.section` directive as follows: `    .section .note.GNU-stack, "x"` The command-line option `-Wa,--noexecstack` overrides the use of the `.section` directive.

Table 5-1 Comparison of command-line options in `armasm` and the `armclang` integrated assembler (continued)

`armasm` option	`armclang` integrated assembler option	Description
`--keep`	No direct equivalent.	With `armasm`, the option instructs the assembler to keep named local labels in the symbol table of the object file, for use by the debugger. With the `armclang` integrated assembler, named local labels defined without using the GNU assembly local symbol name prefix `.L` are always preserved in the object file. Use the command-line option `-Wa,-L` to automatically preserve all named local labels defined using the GNU assembly local symbol name prefix.
`-M`	`-M`	Instructs the assembler to produce a list of makefile dependency lines suitable for use by a make utility. **Note** Only dependencies visible to the preprocessor are included. Files added using the GNU assembler syntax `.incbin` or `.include` directives (or armasm syntax `INCBIN`, `INCLUDE`, or `GET` directives) are not included. **Note** With the `armclang` integrated assembler, using this option with `-o` outputs the makefile dependency lines to the file specified. An object file is not produced.
`--mm`	`-MM`	Creates a single makefile dependency file, without the system header files. **Note** Only dependencies visible to the preprocessor are included. Files added using the GNU assembler syntax `.incbin` or `.include` directives (or armasm syntax `INCBIN`, `INCLUDE`, or `GET` directives) are not included. **Note** With the `armclang` integrated assembler, using this option with `-o` outputs the makefile dependency file to the file specified. An object file is not produced.
`--no_hide_all`	`-fvisibility=default`	Gives all exported and imported global symbols `STV_DEFAULT` visibility in `ELF` rather than `STV_HIDDEN`, unless overridden using source directives.

Table 5-1 Comparison of command-line options in `armasm` and the `armclang` integrated assembler (continued)

`armasm` option	`armclang` integrated assembler option	Description
`--predefine "`*`directive`*`"`, `--pd "`*`directive`*`"`	`-Wa,-defsym,`*`symbol`*`=`*`value`*	With `armasm`, the option instructs the assembler to pre-execute one of the `SETA`, `SETL`, or `SETS` directives as specified using *`directive`*. With the `armclang` integrated assembler, the option instructs the assembler to pre-define the symbol *`symbol`* with the value *`value`*. This GNU assembly `.set` directive can be used to change this value in the file being assembled.

`--unaligned_access`,
`--no_unaligned_access`
	No direct equivalent.	With `armasm`, the options instruct the assembler to set an attribute in the object file to enable or disable the use of unaligned accesses.
		With the `armclang` integrated assembler, use the GNU assembly `.eabi_attribute` directive instead.
		To enable the use of unaligned access, use the directive as follows:
		`    .eabi_attribute Tag_CPU_unaligned_access, 1`
		To disable the use of unaligned access, use the directive as follows:
		`    .eabi_attribute Tag_CPU_unaligned_access, 0`

`--unsafe`	No direct equivalent.	With `armasm`, the option enables instructions for architectures other than the target architecture to be assembled without error. With the `armclang` integrated assembler, use the GNU assembly `.inst` directive to generate such instructions.

Related information

GNU Binutils - Using as: .section

GNU Binutils - Using as: .ident

GNU Binutils - Using as: .protected

GNU Binutils - Using as: ARM Machine Directives

GNU Binutils - Using as: .include

GNU Binutils - Using as: .incbin

GNU Binutils - Using as: Symbol Names

GNU Binutils - Using as: .set

Reference Guide: GET or INCLUDE

Reference Guide: INCBIN

Reference Guide: -mimplicit-it

5.2 Overview of differences between armasm and GNU syntax assembly code

`armasm` (for assembling legacy assembly code) uses `armasm` syntax assembly code.

`armclang` aims to be compatible with GNU syntax assembly code (that is, the assembly code syntax supported by the GNU assembler, `as`).

If you have legacy assembly code that you want to assemble with `armclang`, you must convert that assembly code from `armasm` syntax to GNU syntax.

The specific instructions and order of operands in your UAL syntax assembly code do not change during this migration process.

However, you need to make changes to the syntax of your assembly code. These changes include:

- The directives in your code.
- The format of labels, comments, and some types of literals.
- Some symbol names.
- The operators in your code.

The following examples show simple, equivalent, assembly code in both `armasm` and GNU syntax.

armasm syntax

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

        AREA ||.text||, CODE, READONLY, ALIGN=2

main PROC
        MOV      w5,#0x64      ; W5 = 100
        MOV      w4,#0         ; W4 = 0
        B        test_loop     ; branch to test_loop
loop
        ADD      w5,w5,#1      ; Add 1 to W5
        ADD      w4,w4,#1      ; Add 1 to W4
test_loop
        CMP      w4,#0xa       ; if W4 < 10, branch back to loop
        BLT      loop
        ENDP

        END
```

GNU syntax

```
// Simple GNU syntax example 5.3 Comments on page 5-83//
// Iterate round a loop 10 times, adding 1 to a register each time.

        .section .text,"ax"    // 5.7 Sections on page 5-88
        .balign 4

main:                          // 5.4 Labels on page 5-84
        MOV      w5,#0x64      // W5 = 100  5.9 Numeric literals on page 5-91
        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                   // 5.18 Miscellaneous directives on page 5-102
```

Related references

5.3 Comments on page 5-83
5.4 Labels on page 5-84
5.5 Numeric local labels on page 5-85
5.6 Functions on page 5-87
5.7 Sections on page 5-88
5.8 Symbol naming rules on page 5-90

5.9 Numeric literals on page 5-91
5.10 Operators on page 5-92
5.11 Alignment on page 5-93
5.12 PC-relative addressing on page 5-94
5.15 Conditional directives on page 5-98
5.16 Data definition directives on page 5-99
5.17 Instruction set directives on page 5-101
5.18 Miscellaneous directives on page 5-102
5.19 Symbol definition directives on page 5-104

Related information

About the Unified Assembler Language

5.3 Comments

A comment identifies text that the assembler ignores.

armasm syntax

A comment is the final part of a source line. The first semicolon on a line marks the beginning of a comment except where the semicolon appears inside a string literal.

The end of the line is the end of the comment. A comment alone is a valid line.

For example:

```
; This whole line is a comment
; And also this line

myProc: PROC
    MOV     r1, #16    ; Load R0 with 16
```

GNU syntax

GNU syntax assembly code provides two different methods for marking comments:

- The `/*` and `*/` markers identify multiline comments:

  ```
  /* This is a comment
  that spans multiple
  lines */
  ```

- The `//` marker identifies the remainder of a line as a comment:

  ```
  MOV R0,#16    // Load R0 with 16
  ```

Related information

GNU Binutils - Using as: Comments

Syntax of source lines in assembly language

5.4 Labels

Labels are symbolic representations of addresses. You can use labels to mark specific addresses that you want to refer to from other parts of the code.

armasm syntax

A label is written as a symbol beginning in the first column. A label can appear either in a line on its own, or in a line with an instruction or directive. Whitespace separates the label from any following instruction or directive:

```
  MOV R0,#16
loop SUB R0,R0,#1 ; "loop" is a label
  CMP R0,#0
  BGT loop
```

GNU syntax

A label is written as a symbol that either begins in the first column, or has nothing but whitespace between the first column and the label. A label can appear either in a line on its own, or in a line with an instruction or directive. A colon ":" follows the label (whitespace is allowed between the label and the colon):

```
  MOV R0,#16
loop:              // "loop" label on its own line
  SUB R0,R0,#1
  CMP R0,#0
  BGT loop
```

```
  MOV R0,#16
loop: SUB R0,R0,#1  // "loop" label in a line with an instruction
  CMP R0,#0
  BGT loop
```

Related references

5.5 Numeric local labels on page 5-85

Related information

GNU Binutils - Using as: Labels

5.5 Numeric local labels

Numeric local labels are a type of label that you refer to by a number rather than by name. Unlike other labels, the same numeric local label can be used multiple times and the same number can be used for more than one numeric local label.

armasm syntax

A numeric local label is a number in the range 0-99, optionally followed by a scope name corresponding to a `ROUT` directive.

Numeric local labels follow the same syntax as all other labels.

Refer to numeric local labels using the following syntax:

`%[F|B][A|T]n[routname]`

Where:

- `F` and `B` instruct the assembler to search forwards and backwards respectively. By default, the assembler searches backwards first, then forwards.
- `A` and `T` instruct the assembler to search all macro levels or only the current macro level respectively. By default, the assembler searches all macros from the current level to the top level, but does not search lower level macros.
- *n* is the number of the numeric local label in the range 0-99.
- *routname* is an optional scope label corresponding to a `ROUT` directive. If *routname* is specified in either a label or a reference to a label, the assembler checks it against the name of the nearest preceding `ROUT` directive. If it does not match, the assembler generates an error message and the assembly fails.

For example, the following code implements an incrementing loop:

```
         MOV      r4,#1       ; r4=1
1                             ; Local label
         ADD      r4,r4,#1    ; Increment r4
         CMP      r4,#0x5     ; if r4 < 5...
         BLT      %b1         ; ...branch backwards to local label "1"
```

Here is the same example using a `ROUT` directive to restrict the scope of the local label:

```
routA    ROUT                 ; Start of "routA" scope
         MOV      r4,#1       ; r4=1
1routA                        ; Local label
         ADD      r4,r4,#1    ; Increment r4
         CMP      r4,#0x9     ; if r4 < 9...
         BLT      %b1routA    ; ...branch backwards to local label "1routA"
routB    ROUT                 ; Start of "routB" scope (and therefore end of "routA" scope)
```

GNU syntax

A numeric local label is a number in the range 0-99.

Numeric local labels follow the same syntax as all other labels.

Refer to numeric local labels using the following syntax:

`n{f|b}`

Where:

- *n* is the number of the numeric local label in the range 0-99.
- `f` and `b` instruct the assembler to search forwards and backwards respectively. There is no default. You must specify one of `f` or `b`.

For example, the following code implements an incrementing loop:

```
         MOV      r4,#1       // r4=1
1:                            // Local label
         ADD      r4,r4,#1    // Increment r4
```

```
        CMP      r4,#0x5     // if r4 < 5...
        BLT      1b          // ...branch backwards to local label "1"
```

Note

GNU syntax assembly code does not provide mechanisms for restricting the scope of local labels.

Related references

5.4 Labels on page 5-84

Related information

GNU Binutils - Using as: Labels

GNU Binutils - Using as: Local labels

Reference Guide: Labels

Reference Guide: Numeric local labels

Reference Guide: Syntax of numeric local labels

Reference Guide: ROUT

5.6 Functions

Assemblers can identify the start of a function when producing DWARF call frame information for ELF.

armasm syntax

The `FUNCTION` directive marks the start of a function. `PROC` is a synonym for `FUNCTION`.

The `ENDFUNC` directive marks the end of a function. `ENDP` is a synonym for `ENDFUNC`.

For example:

```
myproc  PROC
  ; Procedure body
  ENDP
```

GNU syntax

Use the `.type` directive to identify symbols as functions. For example:

```
  .type myproc, "function"
myproc:
  // Procedure body
```

GNU syntax assembly code provides the `.func` and `.endfunc` directives. However, these are not supported by `armclang`. `armclang` uses the `.size` directive to set the symbol size:

```
  .type myproc, "function"
myproc:
  // Procedure body
.Lmyproc_end0:
  .size myproc, .Lmyproc_end0-myproc
```

Note

Functions must be typed to link properly.

Related information

GNU Binutils - Using as: .type

Reference Guide: FUNCTION or PROC

Reference Guide: ENDFUNC or ENDP

5.7 Sections

Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.

armasm syntax

The `AREA` directive instructs the assembler to assemble a new code or data section.

Section attributes within the `AREA` directive provide information about the section. Available section attributes include the following:

- `CODE` specifies that the section contains machine instructions.
- `READONLY` specifies that the section must not be written to.
- `ALIGN=`*n* specifies that the section is aligned on a 2^n byte boundary

For example:

```
AREA mysection, CODE, READONLY, ALIGN=3
```

Note

The `ALIGN` attribute does not take the same values as the `ALIGN` directive. `ALIGN=`*n* (the `AREA` attribute) aligns on a 2^n byte boundary. `ALIGN` *n* (the `ALIGN` directive) aligns on an *n*-byte boundary.

GNU syntax

The `.section` directive instructs the assembler to assemble a new code or data section.

Flags provide information about the section. Available section flags include the following:

- `a` specifies that the section is allocatable.
- `x` specifies that the section is executable.
- `w` specifies that the section is writable.
- `S` specifies that the section contains null-terminated strings.

For example:

```
.section  mysection,"ax"
```

Not all `armasm` syntax `AREA` attributes map onto GNU syntax `.section` flags. For example, the `armasm` syntax `ALIGN` attribute corresponds to the GNU syntax `.balign` directive, rather than a `.section` flag:

```
.section  mysection,"ax"
.balign 8
```

Note

When using Arm Compiler 5, section names do not need to be unique. Therefore, you could use the same section name to create different section types.

Arm Compiler 6 supports multiple sections with the same section name only if you specify a *unique ID*. You must ensure that different section types either:

- Have a unique section name.
- Have a unique ID, if they have the same section name.

If you use the same section name for another section or symbol, without a unique ID, then `armclang` integrated assembler merges the sections and gives the merged section the flags of the first section with that name.

```
// stores both the code and data in one section
// uses the flags from the first section
  .section "sectionX", "ax"
  mov r0, r0
  .section "sectionX", "a", %progbits
  .word 0xdeadbeef
```

```
// stores both the code and data in one section
// uses the flags from the first section
  .section "sectionY", "a", %progbits
  .word 0xdeadbeef
  .section "sectionY", "ax"
  mov r0, r0
```

When you assemble the above example code with:

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

The armclang integrated assembler:

- merges the two sections named `sectionX` into one section with the flags `"ax"`.
- merges the two sections named `sectionY` into one section with the flags `"a", %progbits`.

Related information

GNU Binutils - Using as: .section

GNU Binutils - Using as: .align

Reference Guide: AREA

5.8 Symbol naming rules

`armasm` syntax assembly code and GNU syntax assembly code use similar, but different naming rules for symbols.

Symbol naming rules which are common to both `armasm` syntax and GNU syntax include:

- Symbol names must be unique within their scope.
- Symbol names are case-sensitive, and all characters in the symbol name are significant.
- Symbols must not use the same name as built-in variable names or predefined symbol names.

Symbol naming rules which differ between `armasm` syntax and GNU syntax include:

- `armasm` syntax symbols must start with a letter or the underscore character "_".

  GNU syntax symbols must start with a letter, the underscore character "_", or a period ".".
- `armasm` syntax symbols use double bars to delimit symbol names containing non-alphanumeric characters (except for the underscore):

  ```
  IMPORT ||Image$$ARM_LIB_STACKHEAP$$ZI$$Limit||
  ```

  GNU syntax symbols do not require double bars:

  ```
  .global Image$$ARM_LIB_STACKHEAP$$ZI$$Limit
  ```

Related information

GNU Binutils - Using as: Symbol Names

Reference Guide: Symbol naming rules

5.9 Numeric literals

`armasm` syntax assembly and GNU syntax assembly provide different methods for specifying some types of numeric literal.

Implicit shift operations

`armasm` syntax assembly allows immediate values with an implicit shift operation. For example, the `MOVK` instruction takes a 16-bit operand with an optional left shift. `armasm` accepts the instruction `MOVK x1, #0x40000`, converting the operand automatically to `MOVK x1, #0x4, LSL #16`.

GNU syntax assembly expects immediate values to be presented as encoded. The instruction `MOVK x1, #0x40000` results in the following message: `error: immediate must be an integer in range [0, 65535]`.

Hexadecimal literals

`armasm` syntax assembly provides two methods for specifying hexadecimal literals, the prefixes "&" and "0x".

For example, the following are equivalent:

```
ADD     r1, #0xAF
ADD     r1, #&AF
```

GNU syntax assembly only supports the "0x" prefix for specifying hexadecimal literals. Convert any "&" prefixes to "0x".

n_base-n-digits format

`armasm` syntax assembly lets you specify numeric literals using the following format:

`n_base-n-digits`

For example:
- `2_1101` is the binary literal 1101 (13 in decimal).
- `8_27` is the octal literal 27 (23 in decimal).

GNU syntax assembly does not support the `n_base-n-digits` format. Convert all instances to a supported numeric literal form.

For example, you could convert:

```
ADD     r1, #2_1101
```

to:

```
ADD     r1, #13
```

or:

```
ADD     r1, #0xD
```

Related information

GNU Binutils - Using as: Integers

Reference Guide: Syntax of numeric literals

5.10 Operators

`armasm` syntax assembly and GNU syntax assembly provide different methods for specifying some operators.

The following table shows how to translate `armasm` syntax operators to GNU syntax operators.

Table 5-2 Operator translation

`armasm` syntax operator	GNU syntax operator
`:OR:`	`|`
`:EOR:`	`^`
`:AND:`	`&`
`:NOT:`	`~`
`:SHL:`	`<<`
`:SHR:`	`>>`
`:LOR:`	`||`
`:LAND:`	`&&`
`:ROL:`	No GNU equivalent
`:ROR:`	No GNU equivalent

Related information

GNU Binutils - Using as: Infix Operators

Reference Guide: Unary operators

Reference Guide: Shift operators

Reference Guide: Addition, subtraction, and logical operators

5.11 Alignment

Data and code must be aligned to appropriate boundaries.

For example, The T32 pseudo-instruction `ADR` can only load addresses that are word aligned, but a label within T32 code might not be word aligned. You must use an alignment directive to ensure four-byte alignment of an address within T32 code.

An alignment directive aligns the current location to a specified boundary by padding with zeros or `NOP` instructions.

Note

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

armasm syntax

`armasm` syntax assembly provides the `ALIGN` *n* directive, where *n* specifies the alignment boundary in bytes. For example, the directive `ALIGN 128` aligns addresses to 128-byte boundaries.

`armasm` syntax assembly also provides the `PRESERVE8` directive. The `PRESERVE8` directive specifies that the current file preserves eight-byte alignment of the stack.

GNU syntax

GNU syntax assembly provides the `.balign` *n* directive, which uses the same format as `ALIGN`.

Convert all instances of `ALIGN` *n* to `.balign` *n*.

Note

GNU syntax assembly also provides the `.align` *n* directive. However, the format of *n* varies from system to system. The `.balign` directive provides the same alignment functionality as `.align` with a consistent behavior across all architectures.

Convert all instances of `PRESERVE8` to `.eabi_attribute Tag_ABI_align_preserved, 1`.

Related information

GNU Binutils - Using as: ARM Machine Directives
GNU Binutils - Using as: .align
GNU Binutils - Using as: .balign
Reference Guide: REQUIRE8 and PRESERVE8
Reference Guide: ALIGN

5.12 PC-relative addressing

`armasm` syntax assembly and GNU syntax assembly provide different methods for performing PC-relative addressing.

armasm syntax

`armasm` syntax assembly provides the symbol `{pc}` to let you specify an address relative to the current instruction.

For example:

```
ADRP x0, {pc}
```

GNU syntax

GNU syntax assembly does not support the `{pc}` symbol. Instead, it uses the special dot "." character, as follows:

```
ADRP x0, .
```

Related information

GNU Binutils - Using as: The Special Dot Symbol

Reference Guide: Register-relative and PC-relative expressions

5.13 A32 and T32 instruction substitutions

In certain circumstances, if the value of an Operand2 constant is not available with a given instruction, but its logical inverse or negation is available, then `armasm` can produce an equivalent instruction with the inverted or negated constant. The `armclang` integrated assembler provides limited support for such substitutions.

Substitutions when using `armasm`

More information about the *syntax of Operand2 constants* is available in the *Instruction Set Assembly Guide*. The following table shows the instruction substitutions supported by `armasm`, based on the values of Operand2 constants for the A32 and T32 instruction sets. The equivalent instructions shown can be used manually with the `armclang` integrated assembler for instructions where automatic substitution is not supported.

Table 5-3 A32 and T32 instruction substitutions supported by `armasm`

A32 and T32 instruction	Equivalent instruction	Constant substitution method
`ADC{S}{cond} {Rd}, Rn, #constant`	`SBC{S}{cond} {Rd}, Rn, #~constant`	Logical inversion
`ADD{S}{cond} {Rd}, Rn, #constant`	`SUB{S}{cond} {Rd}, Rn, #-constant`	Negation
`AND{S}{cond} Rd, Rn, #constant`	`BIC{S}{cond} Rd, Rn, #~constant`	Logical inversion
`BIC{S}{cond} Rd, Rn, #constant`	`AND{S}{cond} Rd, Rn, #~constant`	Logical inversion
`CMP{cond} Rn, #constant`	`CMN{cond} Rn, #-constant`	Negation
`CMN{cond} Rn, #constant`	`CMP{cond} Rn, #-constant`	Negation
`MOV{S}{cond} Rd, #constant`	`MVN{S}{cond} Rd, #~constant`	Logical inversion
`MVN{S}{cond} Rd, #constant`	`MOV{S}{cond} Rd, #~constant`	Logical inversion
`ORN{S}{cond} Rd, Rn, #constant` (T32 only)	`ORR{S}{cond} Rd, Rn, #~constant` (T32 only)	Logical inversion
`ORR{S}{cond} Rd, Rn, #constant` (T32 only)	`ORN{S}{cond} Rd, Rn, #~constant` (T32 only)	Logical inversion
`SBC{S}{cond} {Rd}, Rn, #constant`	`ADC{S}{cond} {Rd}, Rn, #~constant`	Logical inversion
`SUB{S}{cond} {Rd}, Rn, #constant`	`ADD{S}{cond} {Rd}, Rn, #-constant`	Negation

To find instruction substitutions in code assembled using `armasm`, use the command-line option `--diag_warning=1645`.

Substitutions when using `armclang` integrated assembler

The `armclang` integrated assembler is also able to produce valid equivalent instructions through substitution, by inverting or negating the specified immediate value. This applies to both assembly language source files and to inline assembly code in C and C++ language source files.

You can disable this substitution using the `-mno-neg-immediates` `armclang` option.

Related information

-mno-neg-immediates armclang option
Syntax of Operand2 as a constant
ADC
ADD

AND
BIC
CMP and CMN
MOV
MVN
ORN
ORR
SBC
SUB

5.14 A32 and T32 pseudo-instructions

`armasm` supports several A32 and T32 pseudo-instructions. The support for the pseudo-instructions varies with the `armclang` integrated assembler.

More information about the A32 and T32 pseudo-instructions is available in the *Arm® Compiler Reference Guide*. The following table shows how to migrate the pseudo-instructions for use with the `armclang` integrated assembler:

Table 5-4 A32 and T32 pseudo-instruction migration

A32 and T32 pseudo-instruction	**`armclang` integrated assembler equivalent**
`ADRL{cond} Rd, label`	No equivalent. Use an `ADR` instruction if *`label`* is within the supported offset range. Use an `LDR` pseudo-instruction if *`label`* is outside the supported offset range for an `ADR` instruction.
`CPY{cond} Rd, Rm`	`mov{cond} Rd, Rm`
`LDR{cond}{.W} Rt, =expr`	Identical.
`LDR{cond}{.W} Rt, =label_expr`	Identical.

`MOV32{cond} Rd, expr`	Use the following instruction sequence:
	`movw{cond} Rd, #:lower16:expr`
	`movt{cond} Rd, #:upper16:expr`

`NEG{cond} Rd, Rm`	`rsbs{cond} Rd, Rm, #0`

`UND{cond}{.W} {#expr}`	Use the following instruction for the A32 instruction set:
	`udf{c}{q} {#}imm`
	Use the following instruction for the T32 instruction set with 8-bit encoding:
	`udf{c}{q} {#}imm`
	Use the following instruction for the T32 instruction set with 16-bit encoding:
	`udf{c}.w  {#}imm`

Related information

Reference Guide: ADRL pseudo-instruction
Reference Guide: CPY pseudo-instruction
Reference Guide: LDR pseudo-instruction
MOV
Reference Guide: MOV32 pseudo-instruction
MOVT
Reference Guide: NEG pseudo-instruction
RSB
UDF
Reference Guide: UND pseudo-instruction

5.15 Conditional directives

Conditional directives specify conditions that control whether or not to assemble a sequence of assembly code.

The following table shows how to translate `armasm` syntax conditional directives to GNU syntax directives:

Table 5-5 Conditional directive translation

`armasm` syntax directive	GNU syntax directive
`IF`	`.if` family of directives
`IF :DEF:`	`.ifdef`
`IF :LNOT::DEF:`	`.ifndef`
`ELSE`	`.else`
`ELSEIF`	`.elseif`
`ENDIF`	`.endif`

In addition to the change in directives shown, the following syntax differences apply:

- In `armasm` syntax, the conditional directives can use forward references. This is possible as `armasm` is a two-pass assembler. In GNU syntax, forward references are not supported, as the `armclang` integrated assembler only performs one pass over the main text.

  If a forward reference is used with the `.ifdef` directive, the condition will always fail implicitly. Similarly, if a forward reference is used with the `.ifndef` directive, the condition will always pass implicitly.
- In `armasm` syntax, the maximum total nesting depth for directive structures such as `IF...ELSE...ENDIF` is 256. In GNU syntax, this limit is not applicable.

Related information

GNU Binutils - Using as: .if

GNU Binutils - Using as: .else

GNU Binutils - Using as: .elseif

GNU Binutils - Using as: .endif

Reference Guide: IF, ELSE, ENDIF, and ELIF

5.16 Data definition directives

Data definition directives allocate memory, define data structures, and set initial contents of memory.

The following table shows how to translate `armasm` syntax data definition directives to GNU syntax directives:

Note

This list only contains examples of common data definition assembly directives. It is not exhaustive.

Table 5-6 Data definition directives translation

`armasm` syntax directive	GNU syntax directive	Description
`DCB`	`.byte`	Allocate one-byte blocks of memory, and specify the initial contents.
`DCW`	`.hword`	Allocate two-byte blocks of memory, and specify the initial contents.
`DCD`	`.word`	Allocate four-byte blocks of memory, and specify the initial contents.
`DCI`	`.inst`	Allocate a block of memory in the code, and specify the opcode. In A32 code, this is a four-byte block. In T32 code, this can be a two-byte or four-byte block. `.inst.n` allocates a two-byte block and `.inst.w` allocates a four-byte block.
`DCQ`	`.quad`	Allocate eight-byte blocks of memory, and specify the initial contents.
`SPACE`	`.org`	Allocate a zeroed block of memory. The `armasm` syntax `SPACE` directive allocates a zeroed block of memory with the specified size. The GNU assembly `.org` directive zeroes the memory up to the given address. The address must be greater than the address at which the directive is placed. The following example shows the `armasm` syntax and GNU syntax methods of creating a 100-byte zeroed block of memory using these directives:

```
;  armasm syntax implementation
start_address   SPACE   0x100

// GNU syntax implementation
start_address:
.org    start_address + 0x100
```

Note

If label arithmetic is not required, the GNU assembly `.space` directive can be used instead of the `.org` directive. However, Arm recommends using the `.org` directive wherever possible.

The following examples show how to rewrite a vector table in both `armasm` and GNU syntax.

`armasm` syntax

```
Vectors
LDR PC, Reset_Addr
LDR PC, Undefined_Addr
LDR PC, SVC_Addr
LDR PC, Prefetch_Addr
LDR PC, Abort_Addr
B .                       ; Reserved vector
LDR PC, IRQ_Addr
LDR PC, FIQ_Addr

Reset_Addr      DCD     Reset_Handler
Undefined_Addr  DCD     Undefined_Handler
SVC_Addr        DCD     SVC_Handler
Prefetch_Addr   DCD     Prefetch_Handler
Abort_Addr      DCD     Abort_Handler
IRQ_Addr        DCD     IRQ_Handler
FIQ_Addr        DCD     FIQ_Handler
```

GNU syntax

```
Vectors:
ldr pc, Reset_Addr
ldr pc, Undefined_Addr
ldr pc, SVC_Addr
ldr pc, Prefetch_Addr
ldr pc, Abort_Addr
b .                       // Reserved vector
ldr pc, IRQ_Addr
ldr pc, FIQ_Addr

.balign 4
Reset_Addr:
.word Reset_Handler
Undefined_Addr:
.word Undefined_Handler
SVC_Addr:
.word SVC_Handler
Prefetch_Addr:
.word Prefetch_Handler
Abort_Addr:
.word Abort_Handler
IRQ_Addr:
.word IRQ_Handler
FIQ_Addr:
word FIQ_Handler
```

Related information

GNU Binutils - Using as: .byte

GNU Binutils - Using as: .word

GNU Binutils - Using as: .hword

GNU Binutils - Using as: .quad

GNU Binutils - Using as: .space

GNU Binutils - Using as: .org

GNU Binutils - Using as: ARM Machine Directives

5.17 Instruction set directives

Instruction set directives instruct the assembler to interpret subsequent instructions as either A32 or T32 instructions.

The following table shows how to translate `armasm` syntax instruction set directives to GNU syntax directives:

Table 5-7 Instruction set directives translation

armasm syntax directive	GNU syntax directive	Description
`ARM` or `CODE32`	`.arm` or `.code 32`	Interpret subsequent instructions as A32 instructions.
`THUMB` or `CODE16`	`.thumb` or `.code 16`	Interpret subsequent instructions as T32 instructions.

Related information

GNU Binutils - Using as: ARM Machine Directives

Reference Guide: ARM or CODE32 directive

Reference Guide: CODE16 directive

Reference Guide: THUMB directive

5.18 Miscellaneous directives

Miscellaneous directives perform a range of different functions.

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page 1-14.

The following table shows how to translate `armasm` syntax miscellaneous directives to GNU syntax directives:

Table 5-8 Miscellaneous directives translation

`armasm` syntax directive	GNU syntax directive	Description
`foo EQU 0x1C`	`.equ foo, 0x1C`	Assigns a value to a symbol. Note the rearrangement of operands. `.equ` is a synonym for `.set`.

`EXPORT StartHere`
`GLOBAL StartHere`
	`.global StartHere`
	`.type StartHere, @function`
		Declares a symbol that can be used by the linker (that is, a symbol that is visible to the linker).
		`armasm` automatically determines the types of exported symbols. However, `armclang` requires that you explicitly specify the types of exported symbols using the `.type` directive.
		If the `.type` directive is not specified, the linker outputs warnings of the form:
		`Warning: L6437W: Relocation #RELA:1 in test.o(.text) with respect to` *`symbol`*`...`
		`Warning: L6318W: test.o(.text) contains branch to a non-code symbol` *`symbol`*`.`

`GET file` `INCLUDE file`	`.include file`	Includes a file within the file being assembled.
`IMPORT foo`	`.global foo`	Provides the assembler with a name that is not defined in the current assembly.
`INCBIN`	`.incbin`	Includes a file within the file being assembled. The file is included verbatim. The assembler always emits a `$d` (data) mapping symbol for the `.incbin` directive.[COMMUNITY]
`INFO` *`n`*`, "string"`	`.warning "string"`	The `INFO` directive supports diagnostic generation on either pass of the assembly (specified by *`n`*). The `.warning` directive does not let you specify a particular pass, because the `armclang` integrated assembler only performs one pass.
`ENTRY`	`armlink --entry=`*`location`*	The `ENTRY` directive declares an entry point to a program. `armclang` does not provide an equivalent directive. Use `armlink --entry=`*`location`* to specify the entry point directly to the linker, rather than defining it in the assembly code.

Table 5-8 Miscellaneous directives translation (continued)

`armasm` syntax directive	GNU syntax directive	Description
`END`	`.end`	Marks the end of the assembly file.
`PRESERVE8`	`.eabi_attribute Tag_ABI_align_preserved, 1`	Emits a build attribute which guarantees that the functions in the file preserve 8-byte stack alignment. **Note** For `armasm` syntax assembly language source files, even if you do not specify the `PRESERVE8` directive, `armasm` automatically emits the build attribute if all functions in the file preserve 8-byte stack alignment. For GNU syntax assembly language source files, the `armclang` integrated assembler does not automatically emit this build attribute. Therefore you must manually inspect and ensure that all functions in your GNU syntax assembly language source file preserve 8-byte stack alignment and then manually add the directive to the file.

Related information

GNU Binutils - Using as: .type
GNU Binutils - Using as: ARM Machine Directives
GNU Binutils - Using as: .warning
GNU Binutils - Using as: .equ
GNU Binutils - Using as: .global
GNU Binutils - Using as: .include
GNU Binutils - Using as: .incbin
Reference Guide: ENTRY
Reference Guide: END
Reference Guide: INFO
Reference Guide: EXPORT or GLOBAL
Reference Guide: --entry

5.19 Symbol definition directives

In `armasm`, symbol definition directives declare and set arithmetic, logical, or string variables. In the GNU assembler syntax, these directives define ELF symbols. There are no direct GNU syntax equivalents for `armasm` variables.

The following table shows how to translate `armasm` syntax symbol definition directives to GNU syntax directives:

Note

This list only contains examples of common symbol definition directives. It is not exhaustive.

Table 5-9 Symbol definition directives translation

`armasm` syntax directive	**GNU syntax directive**	**Description**
`foo RN 11`	`foo .req r11`	Define an alias foo for register R11.
`foo QN q5.I32` `VADD foo, foo, foo`	`foo .req q5` `VADD.I32 foo, foo, foo`	Define an `I32`-typed alias `foo` for the quad-precision register `Q5`. When using the `armasm` syntax, you can specify a typed alias for quad-precision registers. The example defines an `I32`-typed alias `foo` for the quad-precision register `Q5`. When using GNU syntax, you must specify the type on the instruction rather than on the register. The example specifies the `I32` type on the `VADD` instruction.
`foo DN d2.I32` `VADD foo, foo, foo`	`foo .req d2` `VADD.I32 foo, foo, foo`	Define an `I32`-typed alias `foo` for the double-precision register `D2`. When using the `armasm` syntax, you can specify a typed alias for double-precision registers. The example defines an `I32`-typed alias `foo` for the double-precision register `D2`. When using GNU syntax, you must specify the type on the instruction rather than on the register. The example specifies the `I32` type on the `VADD` instruction.

Related information

GNU Binutils - Using as: ARM Machine Directives

GNU Binutils - Using as: .global

GNU Binutils - Using as: .set

5.20 Migration of armasm macros to integrated assembler macros

The `armclang` integrated assembler provides similar macro features to those provided by `armasm`. The macro syntax is based on GNU assembler macro syntax.

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page 1-14.

Note

The following GNU assembly directives are [COMMUNITY] features:

- `.macro` and `.endm`
- `.rept` and `.endr`
- `.error`

Additional information about macro features is available:

- The *Arm® Compiler Reference Guide* provides more detail about the macro directives supported, and examples of using macros.
- The *GNU Binutils - Using as* document provides more detail about GNU assembly macro directives.

Macro directive features

The following table describes the most common `armasm` macro directive features, and shows the equivalent features for the `armclang` integrated assembler.

Table 5-10 Comparison of macro directive features provided by `armasm` and the `armclang` integrated assembler

`armasm` feature	`armclang` integrated assembler feature	Description
`MACRO`, `MEND` directives	`.macro`, `.endm` directives	Directives to mark the start and end of the definition of a macro.
`{$label}` macro parameter	Use a normal macro parameter.	Optionally define an internal label to use within the macro.
`{$cond}` macro parameter	Use a normal macro parameter.	Optionally define a condition code to use within the macro.
`{$parameter{,$parameter}...}` custom macro parameter specification	`{parameter{:type}{,parameter{:type}}...}` custom macro parameter and parameter type specification	With `armasm`, any number of custom macro parameters can be defined. Unspecified parameters are substituted with an empty string. With the `armclang` integrated assembler, the custom macro parameters can optionally have a parameter type `type`. This can be either `req` or `vararg`. Unspecified parameters are substituted with an empty string. The `req` type specifies a required parameter. The assembler generates an error when instantiating a macro if a required parameter is missing and a default value is not available. The `vararg` type collects all remaining parameters as one parameter. It can only be used as the last parameter within the list of parameters for a given macro. Only one `vararg` parameter can be specified.

Table 5-10 Comparison of macro directive features provided by `armasm` and the `armclang` integrated assembler (continued)

`armasm` feature	**`armclang` integrated assembler feature**	**Description**
`MEXIT` directive	`.exitm` directive	Exit early from a macro definition.
`IF`, `ELSE`, `ELIF`, `ENDIF` conditional assembly directives	`.if` family of directives, and the `.else`, `.elseif`, `.endif` directives	The directives allow conditional assembly of instructions. With `armasm`, the conditional assembly directives use a logical expression that evaluates to either `TRUE` or `FALSE` as their controlling expression. With the `armclang` integrated assembler, multiple variants of the GNU assembly `.if` directive are available, referred to as the `.if` family of directives. For the `.if` and `.elseif` directives, the controlling expression is a logical expression that evaluates to either `TRUE` or `FALSE`. For other directives in the `.if` family of directives, the controlling expression is an implicit part of the directive used, and varies for each such directive.
`WHILE`, `WEND` directives	`.rept`, `.endr` directives	The directives allow a sequence of instructions or directives to be assembled repeatedly. With `armasm`, the `WHILE` directive uses a logical expression that evaluates to either `TRUE` or `FALSE` as its controlling expression. The sequence enclosed between a `WHILE` and `WEND` directive pair is assembled until the logical expression evaluates to `FALSE`. With the `armclang` integrated assembler, the GNU assembly `.rept` directive takes a fixed number of repetitions as a parameter. The sequence enclosed between a `.rept` and `.endr` directive pair is assembled the specified fixed number of times. To replicate the effect of using a logical expression to repeatedly assemble a code sequence, the `.rept` directive can be used within a macro. See the example provided later in this section.
`ASSERT` directive	Use a combination of the `.if` family of directives and the `.error` directive.	With `armasm`, the `ASSERT` directive generates an error message during assembly if a given assertion is false. A logical expression that evaluates to `TRUE` or `FALSE` is used as the assertion. With the `armclang` integrated assembler, this functionality can be achieved by using a GNU assembly directive from the `.if` family of directives to conditionally display an error message during assembly using the GNU assembly `.error` directive. Macros can be created to simplify this process. See the example provided later in this section.

Notable differences between armasm macro syntax and GNU macro syntax

The following syntax restrictions apply to GNU macro syntax in addition to the differences due to macro directives:

- In `armasm` macro syntax, using the pipe character `|` as the parameter value when instantiating a macro selects the default value of the parameter. In GNU macro syntax, leaving the parameter value empty

when instantiating a macro selects the default value of the parameter. If a default value is not specified in the macro definition, an empty string is used.
- In `armasm` macro syntax, a dot can be used between a parameter and subsequent text, or another parameter, if a space is not required in the expansion. In GNU macro syntax, a set of parentheses `()` can be used between a parameter and subsequent text, if a space is not required in the expansion. There is no need to separate a parameter from another subsequent parameter.
- Although the integrated assembler is case-insensitive to register names, the GNU assembly `.ifc` directive always performs a case-sensitive comparison. Manually check that the register names use the same case-sense when comparing them using the directive.

Migration of macro examples provided in the Arm® Compiler Reference Guide

Table 5-11 NOT EQUALS assertion

`armasm` syntax implementation

```
    ASSERT  arg1 <> arg2
```

GNU syntax implementation

```
    /* Helper macro to replicate ASSERT <> directive
       functionality from armasm.
       Displays error if NE assertion fails. */
    .macro  assertNE arg1:req, arg2:req, message:req
    .ifc \arg1, \arg2
        .error "\message"
    .endif
    .endm
```

Table 5-12 Unsigned integer division macro

`armasm` syntax implementation

The macro takes the following parameters:

`$Bot`

The register that holds the divisor.

`$Top`

The register that holds the dividend before the instructions are executed. After the instructions are executed, it holds the remainder.

`$Div`

The register where the quotient of the division is placed. It can be `NULL` (`""`) if only the remainder is required.

`$Temp`

A temporary register used during the calculation.

```
        MACRO
$Lab    DivMod  $Div,$Top,$Bot,$Temp
        ASSERT  $Top <> $Bot        ; Produce an error message if the
        ASSERT  $Top <> $Temp       ; registers supplied are
        ASSERT  $Bot <> $Temp       ; not all different
        IF      "$Div" <> ""
            ASSERT  $Div <> $Top    ; These three only matter if $Div
            ASSERT  $Div <> $Bot    ; is not null ("")
            ASSERT  $Div <> $Temp   ;
        ENDIF
$Lab
        MOV     $Temp, $Bot                 ; Put divisor in $Temp
        CMP     $Temp, $Top, LSR #1         ; double it until
90      MOVLS   $Temp, $Temp, LSL #1        ; 2 * $Temp > $Top
        CMP     $Temp, $Top, LSR #1
        BLS     %b90                        ; The b means search backwards
        IF      "$Div" <> ""                ; Omit next instruction if $Div
                                            ; is null
            MOV     $Div, #0                ; Initialize quotient
        ENDIF
91      CMP     $Top, $Temp                 ; Can we subtract $Temp?
        SUBCS   $Top, $Top,$Temp            ; If we can, do so
        IF      "$Div" <> ""                ; Omit next instruction if $Div
                                            ; is null
            ADC     $Div, $Div, $Div        ; Double $Div
        ENDIF
        MOV     $Temp, $Temp, LSR #1        ; Halve $Temp,
        CMP     $Temp, $Bot                 ; and loop until
        BHS     %b91                        ; less than divisor
        MEND
```

GNU syntax implementation

The macro takes the following parameters:

`Lab`

A label to mark the start of the code. This parameter is required.

`BotRegNum`

The register number for the register that holds the divisor. This parameter is required.

`TopRegNum`

The register number for the register that holds the dividend before the instructions are executed. After the instructions are executed, it holds the remainder. This parameter is required.

`DivRegNum`

The register number for the register where the quotient of the division is placed. It can be `NULL` (`""`) if only the remainder is required. This parameter is optional.

`TempRegNum`

The register number for a temporary register used during the calculation. This parameter is required.

```
    .macro  DivMod Lab:req, DivRegNum, TopRegNum:req, BotRegNum:req, TempRegNum:req
    assertNE \TopRegNum, \BotRegNum, "Top and Bottom cannot be the same register"
    assertNE \TopRegNum, \TempRegNum, "Top and Temp cannot be the same register"
    assertNE \BotRegNum, \TempRegNum, "Bottom and Temp cannot be the same register"
    .ifnb \DivRegNum
        assertNE \DivRegNum, \TopRegNum, "Div and Top cannot be the same register"
        assertNE \DivRegNum, \BotRegNum, "Div and Bottom cannot be the same register"
        assertNE \DivRegNum, \TempRegNum, "Div and Temp cannot be the same register"
    .endif
\Lab:
    mov     r\TempRegNum, r\BotRegNum           // Put divisor in r\TempRegNum
    cmp     r\TempRegNum, r\TopRegNum,  lsr #1  // double it until
90:
    movls   r\TempRegNum, r\TempRegNum, lsl #1  // 2 * r\TempRegNum > r\TopRegNum
    cmp     r\TempRegNum, r\TopRegNum,  lsr #1
    bls     90b                 // The 'b' means search backwards
    .ifnb \DivRegNum            // Omit next instruction if r\DivRegNum is null
        mov r\DivRegNum, #0     // Initialize quotient
    .endif
91:
    cmp     r\TopRegNum, r\TempRegNum               // Can we subtract r\TempRegNum?
    subcs   r\TopRegNum, r\TopRegNum, r\TempRegNum  // If we can, then do so
    .ifnb \DivRegNum            // Omit next instruction if r\DivRegNum is null
        adc r\DivRegNum, r\DivRegNum, r\DivRegNum   // Double r\DivRegNum
    .endif
    mov     r\TempRegNum, r\TempRegNum, lsr #1  // Halve r\TempRegNum
    cmp     r\TempRegNum, r\BotRegNum           // and loop until
    bhs     91b                                 // less than divisor
    .endm
```

Notable differences from the `armasm` syntax implementation:

- A custom macro, `assertNE`, is used instead of the `armasm` `ASSERT` directive.
- Register numbers are used instead of registers as parameters. This is because the GNU assembly `.ifc` directive used for the `assertNE` assertions treats its operands as case-sensitive.
- The GNU assembly `.ifnb` directive is used to check if the parameter `DivRegNum` has been defined. In the `armasm` syntax implementation, the `armasm` `IF` directive is used.

Table 5-13 Assembly-time diagnostics macro

`armasm` syntax implementation

```
        MACRO                        ; Macro definition
        diagnose  $param1="default"  ; This macro produces
        INFO      0,"$param1"        ; assembly-time diagnostics
        MEND                         ; (on second assembly pass)
; macro expansion
        diagnose            ; Prints blank line at assembly-time
        diagnose "hello"    ; Prints "hello" at assembly-time
        diagnose |          ; Prints "default" at assembly-time
```

GNU syntax implementation

```
//  macro definition
    .macro  diagnose, param1="default"
    .warning "\param1"
    .endm
//  macro instantiation
    .section "diagnoseMacro", "ax"
    diagnose ""         // Prints a warning with an empty string at assembly-time
                        // Cannot print blank line as the .print directive is not supported
    diagnose "hello"    // Prints a warning with the message "hello" at assembly-time
    diagnose            // Prints a warning with the default message "default"
                        // at assembly-time
```

Notable differences from the `armasm` syntax implementation:

- It is not possible to print a blank line at assembly-time using the GNU assembly `.warning` directive. Only a warning with an empty message can be printed.
- The format of the diagnostic message displayed is different between `armasm` and the `armclang` integrated assembler.

  With `armasm`, the diagnostic messages displayed at assembly-time by the macro example are:

  ```
  "macros_armasm.S", line 11:
  "macros_armasm.S", line 12: hello
  "macros_armasm.S", line 13: default
  ```

  With the `armclang` integrated assembler, the diagnostic messages displayed at assembly-time by the macro example are:

  ```
  <instantiation>:1:1: warning:
  .warning ""
  ^
  macros_armclang.S:11:5: note: while in macro instantiation
      diagnose ""
      ^
  <instantiation>:1:1: warning: hello
  .warning "hello"
  ^
  macros_armclang.S:13:5: note: while in macro instantiation
      diagnose "hello"
      ^
  <instantiation>:1:1: warning: default
  .warning "default"
  ^
  macros_armclang.S:14:5: note: while in macro instantiation
      diagnose
      ^
  ```

Table 5-14 Conditional loop macro

`armasm` syntax implementation
The macro takes the following parameters: **`$counter`** The assembly-time variable for the loop counter. This parameter is required. The `{$label}` parameter for the `MACRO` directive has been used for this parameter. If a normal macro parameter is used, the parameter cannot be instantiated as a label. **`$N`** The maximum number of iterations for the loop. This parameter is required. **`$decr`** The loop decrement value. This parameter is optional. **`do`** The text to which `$counter` is appended in each iteration of the loop. This parameter is required.

```
    MACRO
$counter    WhileLoop $N, $decr="1", $do  ; macro definition
    ASSERT  "$counter" <> ""              ; check that $counter has been specified
    ASSERT  "$N" <> ""                    ; check that $N has been specified
    ASSERT  "$do" <> ""                   ; check that $do has been specified
    GBLA    $counter                      ; create new local variable $counter
$counter    SETA $N                       ; initialise $counter
    WHILE   $counter > 0                  ; loop while $counter > 0
            $do$counter                   ; assemble in each iteration of the loop
$counter    SETA $counter-$decr           ; decrement the counter by $decr
    WEND
    MEND
; macro instantiation
    AREA    WhileLoopMacro,CODE
    THUMB
counter     WhileLoop 10, 2, "mov r0, #"
    END
```

GNU syntax implementation

The macro takes the following parameters:

`counter`

The assembly-time variable for the loop counter. This parameter is required.

`N`

The maximum number of iterations for the loop. This parameter is required.

`decr`

The loop decrement value. This parameter is optional.

`do`

The text to which `\counter` is appended in each iteration of the loop. This parameter is required.

```
    /* Macro that inserts the \counter value
       at the end of all \do varargs,
       up to N times. */
    .macro WhileLoop, counter:req, N:req, decr=1, do:vararg
    .set \counter, \N    // initialise the variable \counter to 0
    .rept \N             // loop up to \N times
    .ifgt \counter       // assemble only if \counter is greater than zero
        \do\counter
        .set \counter, \counter-\decr    // decrement the counter by \decr
    .endif
    .endr
    .endm
//  macro instantiation
    .section "WhileLoopMacro", "ax"
    WhileLoop counter, 10, 2, mov r0, #
```

Note

The order in which the GNU assembly `.ifgt`, `.endif`, `.rept`, and `.endr` directives are used is important. Including the `.endr` directive as a statement within the `.ifgt ... .endif` structure produces an error. Similarly, placing the `.endif` directive outside the `.rept ... .endr` structure produces an error.

The macro expansion produces the following code:

```
    mov r0, #0xa
    mov r0, #8
    mov r0, #6
    mov r0, #4
    mov r0, #2
```

Notable differences from the `armasm` syntax implementation:

- In the `armasm` syntax implementation, the `ASSERT` directive is used to raise an error if a required parameter is missing. In the GNU syntax implementation, this can be achieved by using the parameter type `req` for required parameters in the macro definition.
- In the `armasm` syntax implementation, the macro instantiation uses a string as the value to the `$do` parameter. The quotes are implicitly removed at assembly-time. Quotes are required as the parameter value contains spaces. In the GNU syntax implementation, this is achieved using the parameter type `vararg` for the `\do` parameter in the macro definition.
- In the GNU syntax implementation, the `.rept ... .endr` structure is always evaluated `\N` times at assembly-time. This is because the `.ifgt ... .endif` structure must be placed within the `.rept ... .endr` structure. In the `armasm` syntax implementation, the `WHILE...WEND` structure is only evaluated the required number of times at assembly-time based on the controlling expression of the `WHILE` directive.

Related information

GNU Binutils - Using as: .error

GNU Binutils - Using as: .macro

GNU Binutils - Using as: .rept
GNU Binutils - Using as: .if
GNU Binutils - Using as: .else
GNU Binutils - Using as: .elseif
GNU Binutils - Using as: .endif
GNU Binutils - Using as: .warning
Reference Guide: ASSERT
Reference Guide: IF, ELSE, ENDIF, and ELIF
Reference Guide: MACRO and MEND
Reference Guide: MEXIT
Reference Guide: WHILE and WEND
Reference Guide: Use of macros
Reference Guide: Unsigned integer division macro example

Chapter 6
Changes Between Different Versions of Arm® Compiler 6

A description of the changes that affect migration and compatibility between different versions of Arm Compiler 6.

Note

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

  You can mix C++ with C code or C libraries.
- All C++ compilation units that are to be linked into a single image must be compiled with the same version of the C++ standard library ABI. If the ABI version changes between Arm Compiler releases, then you must recompile your object files.

  If you are unable to recompile some of your object files, then contact Arm Support at *https://developer.arm.com/support*.

Note

The documentation changes for Arm Compiler 6.15 and later releases are listed in an appendix for each document.

It contains the following sections:

- *6.3 Summary of changes between Arm® Compiler 6.13 and Arm® Compiler 6.14* on page 6-119.
- *6.4 Summary of changes between Arm® Compiler 6.12 and Arm® Compiler 6.13* on page 6-120.
- *6.5 Summary of changes between Arm® Compiler 6.11 and Arm® Compiler 6.12* on page 6-121.
- *6.6 Summary of changes between Arm® Compiler 6.10 and Arm® Compiler 6.11* on page 6-122.
- *6.7 Summary of changes between Arm® Compiler 6.9 and Arm® Compiler 6.10* on page 6-124.
- *6.8 Summary of changes between Arm® Compiler 6.8 and Arm® Compiler 6.9* on page 6-125.
- *6.9 Summary of changes between Arm® Compiler 6.7 and Arm® Compiler 6.8* on page 6-126.
- *6.10 Summary of changes between Arm® Compiler 6.6 and Arm® Compiler 6.7* on page 6-128.
- *6.11 Summary of changes between Arm® Compiler 6.5 and Arm® Compiler 6.6* on page 6-129.
- *6.12 Compiling with -mexecute-only generates an empty .text section* on page 6-132.

6.1 Summary of changes between Arm® Compiler 6.15 and Arm® Compiler 6.16

A summary of the changes between Arm Compiler 6.15 and Arm Compiler 6.16.

Note

This topic includes descriptions of [BETA] features. See *Support level definitions* on page 1-14.

Documentation

The following appendixes list the technical changes in each document:

- *Arm® Compiler User Guide Changes*.
- *Arm® Compiler Reference Guide Changes*.
- *Arm® Compiler Arm® C and C++ Libraries and Floating-Point Support User Guide Changes*.
- *Appendix C Arm® Compiler Migration and Compatibility Guide Changes* on page Appx-C-144.
- *Arm® Compiler Errors and Warnings Reference Guide Changes*.

Architecture and optional extension changes

Armv8.7-A is fully supported, and enables the following in the release:

- The following base Armv8.7-A ISA features:
  - the HCRX_EL2 System register
  - the `WFET`, `WFIT`, `DSBNXS`, and `TLBINXS` instructions.
- The Extended Event Filter of the Statistical Profiling Extension, that consists of an extra register in the Statistical Profiling Extension.
- The Invalidate the Branch Record Buffer extension, `+brbe`.
- The accelerator support extension, `+ls64`, is fully supported. The extension enables the 64-byte load and store instruction family. That is, `LD64B` and `ST64BV0`, and the ACCDATA_EL1 system register.

For more information, see *`-march`* and *`-mcpu`*.

C++ library changes

The following changes have been made to the C++ library, libc++:

- The *Application Binary Interface* (ABI) version used for the C++ library is now version 2. The change of ABI version has some consequences:
  - C++ objects or libraries built using Arm Compiler 6.15 or earlier are not guaranteed to be compatible with C++ objects or libraries built using 6.16. Therefore, you might see link-time errors or, in rare circumstances, unexpected runtime behavior.

    To make sure your C++ objects and libraries are compatible with Arm Compiler 6.16, you must rebuild all your C++ code with 6.16.

    Note

    If you cannot rebuild any of your C++ objects or libraries, then you must continue to use your previous Arm Compiler version.
  - `std::pointer_safety` and `std::get_pointer_safety()` are no longer available in C++03. However, because the `get_pointer_safety()` function always returns `pointer_safety::relaxed`, you can either reimplement the function or avoid using it.
- The Arm Compiler implementation of `std::deque<T>` allocates memory for storing its elements as blocks of certain size. In Arm Compiler 6.15 and earlier versions, the number of elements per block is computed as follows:
  - If `sizeof(T)` < 256, each block can hold `4096/sizeof(T)` elements.
  - Otherwise, each block can hold 16 elements.

  For Arm Compiler 6.16 and later versions, the formula is different:
  - If `sizeof(T)` < 8, each block can hold `64/sizeof(T)` elements.
  - Otherwise, each block can hold 8 elements.

armlink changes

Arm Compiler 5 does not support literal pool merging.

Arm Compiler 6 merges literal pools by default. In Arm Compiler 6.15 and earlier, marking a load region as `PROTECTED` prevents merging of literal pools for **`const`** strings but not for other **`const`** values in literal pools. For example, your code might have three literal pools, each containing the same **`const`** value, where two of the literal pools are in a `PROTECTED` region. In Arm Compiler 6.15 and earlier, `armlink` merges the three **`const`** values but leaves a single copy in the `PROTECTED` regions. However, the function containing the copy still references that copy, but the referencing function is now part of a different region.

In Arm Compiler 6.16, `armlink` prevents merging literal pool entries of **`const`** strings and **`const`** values across regions that have the `PROTECTED` load region attribute.

Here, the terms **`const`** string and **`const`** value have the following meanings:

`const` string

A string literal from an ELF section with the `SHF_MERGE` and `SHF_STRINGS` flags.

`const` value

A constant defined in a constant pool where the constant pool is in the same section as the code that uses it.

6.2 Summary of changes between Arm® Compiler 6.14 and Arm® Compiler 6.15

A summary of the changes between Arm Compiler 6.14 and Arm Compiler 6.15.

Documentation

The following appendixes list the technical changes in each document:

- *Arm® Compiler User Guide Changes*.
- *Arm® Compiler Reference Guide Changes*.
- *Arm® Compiler Arm® C and C++ Libraries and Floating-Point Support User Guide Changes*.
- *Appendix C Arm® Compiler Migration and Compatibility Guide Changes* on page Appx-C-144.
- *Arm® Compiler Errors and Warnings Reference Guide Changes*.

Architecture and optional extension changes

Arm Compiler 6.15 adds the following:

- [ALPHA] support for the Armv8.7-A architecture and the Accelerator Support Extension, `+ls64`, for 64-byte atomic loads and stores.
- [BETA] support for the 64-bit Armv8.7-R architecture.

Other changes are as follows:

- The *Custom Datapath Extension* (CDE), `+cdecpN` is fully supported.

For more information, see *`-march`* and *`-mcpu`*.

Command-line options

Arm Compiler 6.15 adds support for the `armlink` command-line option, `--dangling-debug-address=address`. See *`--dangling-debug-address=address`*.

Arm Compiler 6.15 adds a new `-Omin` command-line option for `armclang` and `armlink` which aims to produce the minimum code size using link-time optimization. The value `Omin` can also be specified for the `armlink` command-line option `--lto-level`. For more information, see the *`-O`* `armclang` reference page.

6.3 Summary of changes between Arm® Compiler 6.13 and Arm® Compiler 6.14

A summary of the changes between Arm Compiler 6.13 and Arm Compiler 6.14.

Note

This topic includes descriptions of [BETA] features. See *Support level definitions* on page 1-14.

New architectures and optional extensions

Arm Compiler 6.14 adds [BETA] support for the *Custom Datapath Extension* (CDE), `+cdecpN`. For more information, see *`-march`*.

New processors

Arm Compiler 6.14 adds Cortex-M55 processor support. For information about the *M-profile Vector Extension* (MVE) and *floating-point* (FP) combinations for this processor, see *Supported architecture feature combinations for specific processors*.

Command-line options

Arm Compiler 6.14 adds [BETA] support for the `fromelf` command-line option, `--coprocN=value`, to enable T32 encodings of the CDE. See *`--coprocN=value`*.

Function attributes

Arm Compiler 6.14 adds support for the `__attribute__((target("options")))` function attribute. See *`__attribute__((target("options")))`* for more information.

Library-related features

The Arm implementation of the C++ standard library class `std::random_device` is described in *Numerics library*.

6.4 Summary of changes between Arm® Compiler 6.12 and Arm® Compiler 6.13

A summary of the changes between Arm Compiler 6.12 and Arm Compiler 6.13.

Note

This topic includes descriptions of [ALPHA] and [BETA] features. See *Support level definitions* on page 1-14.

New architectures and optional extensions

Arm Compiler 6.13 adds:

- Early support for Future Architecture Technologies:
  - Assembly for the Embedded Trace Extension (ETE). This is enabled by default.
  - Assembly for the Trace Buffer Extension (TRBE). This is enabled by default.
  - Assembly for Scalable Vector Extension 2 (SVE2).
  - Assembly and intrinsics for Transactional Memory Extension (TME).

  For more information, see `-march`.
- [ALPHA] support for the Armv8.6-A architecture:
  - [ALPHA] support assembly and intrinsics for the BFloat16 Extension.
  - [ALPHA] support assembly and intrinsics for the Matrix Multiplication Extension.

  For more information, see `-march`.
- Support for the Armv8.1-M architecture:
  - Assembly and intrinsics for the M-profile Vector Extension (MVE).
  - [BETA] support for the automatic vectorization for MVE.

  For more information, see `-march`.
- Intrinsics for the Armv8.5-A architecture Memory Tagging Extension are promoted from [ALPHA] support to full product quality support. See `-march`.

Command-line options

- Arm Compiler 6.13 supports the SysV dynamic linking model, using the following command-line options:
  - `armclang -fpic`, `armclang -fno-pic`
  - `armclang -fsysv`, `armclang -fno-sysv`
  - `armclang -shared`
  - `armlink --dynamiclinker=name`
  - `armlink --import_unresolved`, `armlink --no_import_unresolved`
  - `armlink --soname`
  - `armlink --sysv`
  - `armlink --shared`
- The `armclang -fsanitize=memtag` option replaces the `armclang -mmemtag-stack` option.

6.5 Summary of changes between Arm® Compiler 6.11 and Arm® Compiler 6.12

A summary of the changes between Arm Compiler 6.11 and Arm Compiler 6.12.

Note

This topic includes descriptions of [ALPHA] features. See *Support level definitions* on page 1-14.

New architectures and optional extensions

Arm Compiler 6.12 adds:

- `armclang` inline assembler and integrated assembler support for the Speculation Barrier (SB) instruction in the AArch32 and AArch64 states. This is mandatory for the Armv8.5-A and later architectures. This is optional for the Armv8-A to Armv8.4-A architectures. To enable the use of the SB instruction, use `-march=armv8-a+sb`. For more information, see *`-march`*.
- `armclang` inline assembler and integrated assembler support for the Speculative Store Bypass Safe (SSBS) register and instructions in the AArch64 state. This is mandatory for the Armv8.5-A and later architectures. This is optional for the Armv8-A to Armv8.4-A architectures. To enable the use of the SSBS register and instructions, use `-march=armv8-a+ssbs`. For more information, see *`-march`*.
- `armclang` inline assembler and integrated assembler support for the Prediction Restriction by Context registers and instructions in the AArch64 state. This is mandatory for the Armv8.5-A and later architectures. This is optional for the Armv8-A to Armv8.4-A architectures. To enable the Prediction Restriction by Context registers and instructions, use `-march=armv8-a+predres`. For more information, see *`-march`*.

Command-line options

Arm Compiler 6.12 adds support for the following command-line options.

- These [ALPHA] options support generation of code for protecting the stack with the memory tagging extension:
  - *`armclang -mmemtag-stack`*
  - *`armlink --library_security=v8.5a`*

  The memory tagging extension is optional in Armv8.5-A and later architectures.

  To disable this stack protection, use *`armclang -mno-memtag-stack`*.

  Note

  Arm Compiler 6.12 also adds support for heap protection using the memory tagging extension, when defining the symbol *`__use_memtag_heap`*.

- These options support generation of code for protecting the stack with stack guard variables:
  - *`armclang -fstack-protector`*
  - *`armclang -fstack-protector-strong`*
  - *`armclang -fstack-protector-all`*

  To disable this stack protection, use *`armclang -fno-stack-protector`*.
- The *`armclang -ffixed-r<N>`* option prevents the compiler from using the specified core register, unless the use is required for Arm ABI compliance.

Keywords

Arm Compiler 6.12 adds support for the *`register`* keyword. The *`register`* keyword enables the use of certain core registers as global named register variables in the AArch32 state.

6.6 Summary of changes between Arm® Compiler 6.10 and Arm® Compiler 6.11

A summary of the changes between Arm Compiler 6.10 and Arm Compiler 6.11.

New architectures and optional extensions

Arm Compiler 6.11 adds:

- `armclang` inline assembler and integrated assembler support for the Armv8.5-A architecture. To target the Armv8.5-A architecture, use `-march=armv8.5-a`. For more information, see *`-march`*.
- `armclang` inline assembler and integrated assembler support for the optional Memory Tagging Extension for the Armv8.5-A architecture. To target the Memory Tagging Extension, use `-march=armv8.5-a+memtag`. For more information, see *`-march`*.
- `armclang` inline assembler and integrated assembler support for the optional Random Number Instructions for the Armv8.5-A architecture. To target the Random Number Instructions, use `-march=armv8.5-a+rng`. For more information, see *`-march`*.
- Support for branch protection features for Armv8.3-A and later architectures. For more information, see *`-mbranch-protection`*.
- Support for half-precision floating-point multiply with add or multiply with subtract instructions for Armv8.2-A and later architectures. To target these instructions, use `+fp16fml` with `-mcpu` or `-march`. For more information, see *`-march`* and *`-mcpu`*.

Command-line options

Arm Compiler 6.11 adds support for the following command-line options.

- These options support generation of code with branch protection:
  - *`armclang -mbranch-protection`*
  - *`armlink --library_security`*
- These options control whether the output file contains compiler name and version information:
  - *`armclang -fident`*
  - *`armclang -fno-ident`*
- These options enable the generation of Position Independent eXecute Only (PIXO) library features for Armv7-M targets:
  - *`armclang -mpixolib`*
  - *`armlink --pixolib`*

Deprecated features

Arm Compiler 6.11 deprecates the following features:

- *`__declspec`* has been deprecated.
- Support for ELF sections that contain the legacy `SHF_COMDEF` ELF section flag has been deprecated.
  - The `COMDEF` section attribute of the legacy `armasm` syntax *`AREA`* directive has been deprecated.
  - Linking with legacy objects that contain ELF sections with the legacy `SHF_COMDEF` ELF section flag has been deprecated.
- The legacy R-type dynamic linking model, which does not conform to the *32-bit Application Binary Interface* for the Arm Architecture, has been deprecated.
  - Linking with *`armlink --reloc`* command-line option has been deprecated.
  - Linking without *`armlink --base_platform`*, with a scatter file that contains the *`RELOC`* load region attribute, has been deprecated.

For more information, see *Backwards compatibility issues*.

Removed features

The following options have been removed from Arm Compiler 6.11:

- *`armlink --compress_debug`* and *`--no_compress_debug`* command-line options.
- *`armlink --match=crossmangled`* command-line option.
- *`armlink --strict_enum_size`* and *`--no_strict_enum_size`* command-line options.
- *`armlink --strict_wchar_size`* and *`--no_strict_wchar_size`* command-line options.

Product quality support level

Support for `-std=c++14` and `-std=gnu++14` has changed from [BETA] to fully supported, with the exception of certain C++14 features. For more information, see *`-std`* and *LLVM component versions and language compatibility*.

For earlier versions of the compiler, Arm recommended the use of `-std=c++11` when compiling C++ source files. This recommendation has been removed.

6.7 Summary of changes between Arm® Compiler 6.9 and Arm® Compiler 6.10

A summary of the changes between Arm Compiler 6.9 and Arm Compiler 6.10.

General changes

The following are general changes in Arm Compiler 6.10:

- When using the legacy assembler, `armasm`, to assemble for AArch32 targets that support A32 and T32 instruction sets, the `apcs` interworking default has changed from `/nointerwork` to `/interwork`. If you must use the non-interworking `apcs`, then you must specify `--apcs=/nointerwork` on the command-line of the legacy assembler, `armasm`. However, from Arm Compiler 6.10, the compiler does not include pure A32 libraries for non-interworking `apcs`. Therefore, if you use the non-interworking `apcs` for A32 code and require library support, then `armlink` generates an error unless you provide your own supporting libraries.
- In certain circumstances, when a legacy assembler or linker process invoked the compiler as a subprocess to preprocess a file but all suitable licenses were already in use, the processes could deadlock. This issue has been fixed.
- The default C++ source language mode has changed from gnu++98 to gnu++14.

  gnu++14 language and library features are a [BETA] product feature. Arm recommends compiling with `-std=c++11` to restrict Arm Compiler to using only C++11 language and library features, which are fully supported.

  See *-std* in the *armclang Reference Guide*.

Enhancements

The following are enhancements in Arm Compiler 6.10:

Compiler and integrated assembler (armclang)

Added support for the `-fno-builtin` option that can prevent the compiler from optimizing calls to certain standard C library functions, such as `printf()`. When compiling without `-fno-builtin`, the compiler can replace such calls with inline code or with calls to other library functions.

See *-fno-builtin* in the *armclang Reference Guide*.

6.8 Summary of changes between Arm® Compiler 6.8 and Arm® Compiler 6.9

A summary of the changes between Arm Compiler 6.8 and Arm Compiler 6.9.

General changes

The following are general changes in Arm Compiler 6.9:

- Added support for the Armv8.4-A architecture. To target Armv8.4-A, use the following options:

State	armclang options	armlink, and fromelf options
AArch64	`--target=aarch64-arm-none-eabi` `-march=armv8.4-a`	Do not use the `--cpu=`*`name`* option.
AArch32	`--target=arm-arm-none-eabi` `-march=armv8.4-a`	Do not use the `--cpu=`*`name`* option.

Note

The legacy assembler, `armasm`, does not support the Armv8.4-A architecture.

- Added support for the optional Cryptographic Extension in Armv8.4-A. To target Armv8.4-A with the Cryptographic Extension, use the following options:

State	armclang options	armlink and fromelf options
AArch64	`--target=aarch64-arm-none-eabi` `-march=armv8.4-a+crypto`	Do not use the `--cpu=`*`name`* option.

AArch32	`--target=arm-arm-none-eabi`
	`-march=armv8.4-a`
	`-mfpu=crypto-neon-fp-armv8`
		Do not use the `--cpu=`*`name`* option.

Note

The legacy assembler, `armasm`, does not support the Armv8.4-A architecture.

For more information about selecting specific cryptographic algorithms, see *-mcpu* in the *armclang Reference Guide*.

- A change in Arm Compiler 6.9 means that compiling with `-mexecute-only` always generates an empty `.text` section that is read-only.

  For more information about handling this section, see *6.12 Compiling with -mexecute-only generates an empty .text section* on page 6-132.

6.9 Summary of changes between Arm® Compiler 6.7 and Arm® Compiler 6.8

A summary of the changes between Arm Compiler 6.7 and Arm Compiler 6.8.

General changes

The following are general changes in Arm Compiler 6.8:

- Added support for the optional Dot Product instructions in Armv8.2-A and Armv8.3-A. To target Armv8.2-A and Armv8.3-A with the Dot Product instructions, use the following options:

Processor	armclang options	armasm, armlink, and fromelf options
Armv8.3-A and AArch64 state	`--target=aarch64-arm-none-eabi` `-march=armv8.3-a+dotprod`	`--cpu=8.3-A.64.dotprod`
Armv8.3-A and AArch32 state	`--target=arm-arm-none-eabi` `-march=armv8.3-a+dotprod`	`--cpu=8.3-A.32.dotprod`
Armv8.2-A and AArch64 state	`--target=aarch64-arm-none-eabi` `-march=armv8.2-a+dotprod`	`--cpu=8.2-A.64.dotprod`
Armv8.2-A and AArch32 state	`--target=arm-arm-none-eabi` `-march=armv8.2-a+dotprod`	`--cpu=8.2-A.32.dotprod`

- Added support for the Cortex-A75 and Cortex-A55 processors. To target Cortex-A75 and Cortex-A55, use the following options:

Processor	armclang options	armasm, armlink, and fromelf options
Cortex-A75 for AArch64 state	`--target=aarch64-arm-none-eab` `-mcpu=cortex-a75`	`8.2-A.64`
Cortex-A75 for AArch32 state	`--target=arm-arm-none-eab` `-mcpu=cortex-a75`	`8.2-A.32`

Cortex-A55 for AArch64 state	`--target=aarch64-arm-none-eab`
	`-mcpu=cortex-a55`
	`-mcpu=cortex-a55`
		`8.2-A.64`

Cortex-A55 for AArch32 state	`--target=arm-arm-none-eab`
	`-mcpu=cortex-a55`
	`-mcpu=cortex-a55`
		`8.2-A.32`

- When resolving the relocations of a branch instruction from a function with build attributes that include `~PRES8` to another function with build attributes that include `REQ8`, the linker previously reported:

  ```
  Error: L6238E: <objname>(<secname>) contains invalid call from '~PRES8 (The user
  did not require code to preserve 8-byte alignment of 8-byte data objects)'
  function to 'REQ8 (Code was permitted to depend on the 8-byte alignment of 8-byte
  data items)' function <sym>.
  ```

  This behavior has been changed. By default, the linker no longer reports an error in these circumstances. To restore the previous behavior, use the option `--strict_preserve8_require8`. For more information about this option, see *--strict_preserve8_require8* in the *armlink User Guide*.

  To successfully link with `--strict_preserve8_require8`

  1. Manually inspect assembly language source files that are assembled using the integrated assembler.
  2. Ensure that all functions preserve 8-byte alignment of the stack and of 8-byte data items.
  3. Add the directive `.eabi_attribute Tag_ABI_align_preserved, 1` to each such source file.

Enhancements

The following are enhancements in Arm Compiler 6.8:

Compiler and integrated assembler (armclang)

- Previously, the inline assembler and integrated assembler provided limited support for instruction substitutions for the A32 and T32 instruction sets. Substitution occurs when a valid encoding does not exist for an instruction with a particular immediate, but an equivalent instruction that has the same result with the inverted or negated immediate is available. To disable this feature, use the option `-mno-neg-immediates`.

  When `-mno-neg-immediatess` is not specified, the range of substitutions that the inline assembler and integrated assembler perform has also been extended to cover extra valid substitutions for A64, A32, and T32.

  For more information about this option, see *-mno-neg-immediates* in the *armclang Reference Guide*.
- Added support for:
  - `#pragma clang section`. This pragma enables migration of source code that previously used the legacy `armcc` feature `#pragma arm section`. See *#pragma clang section* in the *armclang Reference Guide*.
  - `-nostdlib` and `-nostdlibinc` options that enable objects to be linked with other ABI-compliant libraries. See *-nostdlib* and *-nostdlibinc* in the *armclang Reference Guide*.
  - `__unaligned` keyword. This keyword aids migration of source code that previously used the legacy `armcc` feature `__packed`. See *__unaligned* in the *armclang Reference Guide*.

General enhancements

Added support for C++14 source language modes. Use one of the following options to enable the compilation of C++14 source code:

- `-std=c++14`.
- `-std=gnu++14`.

See *-std* in the *armclang Reference Guide*.

6.10 Summary of changes between Arm® Compiler 6.6 and Arm® Compiler 6.7

A summary of the changes between Arm Compiler 6.6 and Arm Compiler 6.7.

General changes

The following are general changes in Arm Compiler 6.7:

- Armv8-M architecture-based targets are now supported when using an Arm DS-5 Professional license.
- Arm Compiler 6.7 includes FlexNet Publisher 11.14.1.0 client libraries. This version of the license client is not compatible with previous versions of the FlexNet Publisher license server software. When used with a license server running an `armlmd` or `lmgrd` version earlier than 11.14.1.0, Arm Compiler 6.7 can report any of the following:
  - `Failed to check out a license.Bad message command.`
  - `Failed to check out a license.Version of vendor daemon is too old.`
  - `Flex error code: -83.`
  - `Flex error code: -140.`

  A license server running `armlmd` and `lmgrd` version 11.14.1.0 (or later) is compatible with Arm Compiler 6.7 and all previous releases of Arm tools.

  Arm recommends that you always use the latest version of the license server software that is available from *https://developer.arm.com/products/software-development-tools/license-management/downloads*.
- Previously, when generating execute-only sections, the tools set the ELF section header flag to `SHF_ARM_NOREAD`. For compliance with forthcoming changes to the *Application Binary Interface (ABI) for the Arm® Architecture*, this behavior has changed. For execute-only sections, the tools now set the ELF section header flag to `SHF_ARM_PURECODE`.

Enhancements

The following are enhancements in Arm Compiler 6.7:

Compiler and integrated assembler (armclang)

- Added support for the `-ffp-mode=model` option that you can use to specify the level of floating-point standard compliance:
  - `-ffp-mode=std` selects the default compiler behavior.
  - `-ffp-mode=fast` is equivalent to `-ffast-math`.
  - `-ffp-mode=full` is equivalent to `-fno-fast-math`.

  Arm recommends using `-ffp-mode` rather than `-ffast-math` or `-fno-fast-math`.

  For more information about this option, see *-ffp-mode* in the *armclang Reference Guide*.
- Extended the support for the `__attribute__((value_in_regs))` function attribute to improve compatibility with the equivalent Arm Compiler 5 feature.

  For more information about this attribute, see the *__attribute__((value_in_regs)) function attribute* in the *armclang Reference Guide*.
- Added support for the generation of implicit `IT` blocks when assembling for T32 state. To specify the behavior of the inline assembler and integrated assembler if there are conditional instructions outside `IT` blocks, use the option `-mimplicit-it=`*`name`*.

  For more information about this option, see *-mimplicit-it* in the *armclang Reference Guide*.
- Previously, when compiling at `-Os`, the compiler could over-align literal pools that are generated during vectorization to a 128-bit boundary. This behavior has been changed. The compiler now avoids adding excessive padding.

armlink

Added support for `__at` sections that are named `.bss.ARM.__at_<address>`. The linker places the associated ZI data at the specified address.

6.11 Summary of changes between Arm® Compiler 6.5 and Arm® Compiler 6.6

A summary of the changes between Arm Compiler 6.5 and Arm Compiler 6.6.

General changes

The following are general changes in Arm Compiler 6.6:

- Added support for the Armv8.3-A architecture. To target Armv8.3-A, use the following options:

State	armclang options	armasm, armlink, and fromelf options
AArch64	`--target=aarch64-arm-none-eabi` `-march=armv8.3-a`	`--cpu=8.3-A.64`
AArch32	`--target=arm-arm-none-eabi` `-march=armv8.3-a`	`--cpu=8.3-A.32`

- Added support for the Armv8-A AArch64 state Scalable Vector Extension (SVE) to the compiler. To target bare-metal systems with SVE, use the option `-march=armv8-a+sve`.

  To disassemble objects that have been built for SVE, `llvm-objdump` is provided as an interim solution.

  SVE features are available under a separate license. Contact Arm for more information.
- Added support for the Cortex-R52 processor. To target Cortex-R52, use the following options:

Processor variant	armclang options	armasm, armlink, and fromelf options
D32 and Advanced SIMD	`--target=arm-arm-none-eab` `-mcpu=cortex-r52`	`--cpu=Cortex-R52`
D16 and single-precision only	`--target=arm-arm-none-eab` `-mcpu=cortex-r52 -mfpu=fpv5-d16`	`--cpu=Cortex-R52 --fpu=FPv5-SP`

- Added support for the Cortex-M23 processor. To target Cortex-M23, use the following options:

  armclang

  ```
  --target=arm-arm-none-eabi -mcpu=cortex-m23
  ```

  armasm, armlink, and fromelf

  ```
  --cpu=Cortex-M23
  ```
- Added support for the Cortex-M33 processor. To target Cortex-M33, use the following options:

Processor variant	armclang options	armasm, armlink, and fromelf options
With both DSP and FP	`--target=arm-arm-none-eab` `-mcpu=cortex-m33`	`--cpu=Cortex-M33`
Without DSP but with FP	`--target=arm-arm-none-eab` `-mcpu=cortex-m33+nodsp`	`--cpu=Cortex-M33.no_dsp`

With DSP but without FP	`--target=arm-arm-none-eab`
	`-mcpu=cortex-m33`
	`-mfloat-abi=soft`
		`--cpu=Cortex-M33 --fpu=SoftVFP`

Without both DSP and FP	`--target=arm-arm-none-eab`
	`-mcpu=cortex-m33+nodsp`
	`-mfloat-abi=soft`
		`--cpu=Cortex-M33.no_dsp`
		`--fpu=SoftVFP`

- The default compiler behavior has changed. The following options are selected by default:

- `-fdata-sections`.
- `-ffunction-sections`.
- `-fomit-frame-pointer`.
- `-fvisibility=hidden`.
- Configuration options that select a smaller, less IEEE 754 compliant floating-point math library.

To restore the previous behavior, select from the following options:

- `-fno-data-sections`.
- `-fno-function-sections`.
- `-fno-omit-frame-pointer`.
- `-fvisibility=default` [COMMUNITY].

  Note

  Arm recommends not using this option to restore the previous behavior.

- `-fno-fast-math`.

For more information about support level definitions and a subset of these options, see the *Arm Compiler Reference Guide*.

- The `--cpu=`*`name`* option in `armasm`, `armlink`, and `fromelf` has changed to improve compatibility with the `-mcpu` compiler option.

Replace this option name	With this option name
`Cortex-A5.neon`	`Cortex-A5`
`Cortex-A5.vfp`	`Cortex-A5.no_neon`
`Cortex-A5`	`Cortex-A5.no_neon.no_vfp`
`Cortex-R5F-rev1`	`Cortex-R5`
`Cortex-R5F`	`Cortex-R5-rev0`
`Cortex-R5`	`Cortex-R5-rev0.no_vfp`
`Cortex-R5F-rev1.sp`	`Cortex-R5.sp`
`Cortex-R5-rev1`	`Cortex-R5.no_vfp`
`Cortex-M4F` or `Cortex-M4.fp`	`Cortex-M4`
`Cortex-M4`	`Cortex-M4.no_fp`
`Cortex-M7.fp.dp`	`Cortex-M7`
`Cortex-M7`	`Cortex-M7.no_fp`

- The following linker options are deprecated and are to be removed in a future release:
  - `--compress_debug`.
  - `--gnu_linker_defined_syms`.
  - `--legacyalign`.
  - `--match=crossmangled`.
  - `--strict_enum_size`.
  - `--strict_wchar_size`.

Enhancements

The following are enhancements in Arm Compiler 6.6:

Compiler and integrated assembler (armclang)

Added support for:

- `__attribute__((naked))` function attribute. This function attribute enables migration of Arm Compiler 5 and earlier embedded assembler functions to Arm Compiler 6.
- Use of floating-point code in secure functions when compiling with `-mcmse -mfloat-abi=hard`.

armlink

Added full support for link-time optimization (LTO). To use LTO, specify the `-flto` option to the compiler and the `--lto` option to the linker.

Libraries and system headers

- Added [ALPHA] support for multithreading features in the C++11 standard library, for example `std::atomic` and `std::thread`. The API for these features is in the `arm-tpl.h` header file, but you must implement the low-level interface to the underlying operating system. The specification of this thread porting API is available through a separate document. Contact Arm Support for more information.
- Added support to the Arm C library to implement semihosting calls using the `HLT` instruction for Armv8-A and Armv8-R targets in AArch32 state.
- Added support for use of the C++ library without exceptions. To target C++ without exceptions, compile with the option `-fno-exceptions`.

  When linking objects compiled without exceptions, a specialized C++ library variant is selected that does not have the code-size overhead of exceptions. This C++ library variant has undefined behavior at points where the normal library variant results in an exception being thrown.

6.12 Compiling with -mexecute-only generates an empty .text section

A change between Arm Compiler 6.8 and Arm Compiler 6.9 means that compiling with `-mexecute-only` always generates an empty `.text` section that is read-only. That is, a section that does not have the `SHF_ARM_PURECODE` attribute.

The linker normally removes the empty `.text` section during unused section elimination. However, the unused section elimination does not occur when:

- The image has no entry point.
- You specify one of the following linker options:
  - `--no_remove`.
  - `--keep (<object-file-name>(.text))`.

If you use a scatter file to merge execute-only (XO) and read-only (RO) sections into a single executable region, then the XO sections lose the XO attribute and become RO.

When compiling with `-fno-function-sections`, all functions are placed in the `.text` section with the `SHF_ARM_PURECODE` attribute. As a result, there are two sections with the name `.text`, one with and one without the `SHF_ARM_PURECODE` attribute. You cannot select between the two `.text` sections by name. Therefore, you must use attributes as the selectors in the scatter file to differentiate between XO and RO sections.

Examples

The following example shows how Arm Compiler 6 handles `.text` sections:

1. Create the file `example.c` containing:

```
void foo() {}

int main() {
  foo();
}
```

2. Compile the program and examine the object file with `fromelf`.

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-M3 -mexecute-only -c -o example.o
example.c
fromelf example.o
```

The output shows that `Section #2` is the empty RO `.text` section:

```
...
** Section #1 '.strtab' (SHT_STRTAB)
    Size   : 148 bytes


** Section #2 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 0 bytes (alignment 4)
    Address: 0x00000000


** Section #3 '.text.foo' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_PURECODE]
    Size   : 2 bytes (alignment 4)
    Address: 0x00000000


** Section #4 '.ARM.exidx.text.foo' (SHT_ARM_EXIDX) [SHF_ALLOC + SHF_LINK_ORDER]
    Size   : 8 bytes (alignment 4)
    Address: 0x00000000
    Link to section #3 '.text.foo'


** Section #5 '.rel.ARM.exidx.text.foo' (SHT_REL)
    Size   : 8 bytes (alignment 4)
    Symbol table #13 '.symtab'
    1 relocations applied to section #4 '.ARM.exidx.text.foo'


** Section #6 '.text.main' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_PURECODE]
    Size   : 10 bytes (alignment 4)
    Address: 0x00000000
...
```

3. Create the file `example.scat` containing:

```
LR_XO 0x10000
{
  ER_MAIN_FOO 0x10000
  {
     example.o(.text*)
  }
}

LR_2 0x20000
{
  ER_REST 0x20000
  {
     *(+RO, +ZI)
  }
  ARM_LIB_STACKHEAP 0x80000 EMPTY -0x1000 {}
}
```

4. Create an image file with `armlink` and examine the image file with `fromelf`:

```
armlink --scatter example.scat -o example_scat.axf example.o
fromelf example_scat.axf
```

The output shows that `Section #1` has the `SHF_ARM_PURECODE` attribute:

```
...
** Section #1 'ER_MAIN_FOO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR + SHF_ARM_PURECODE]
    Size   : 16 bytes (alignment 4)
    Address: 0x00010000


** Section #2 'ER_REST' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 604 bytes (alignment 4)
    Address: 0x00020000
...
```

5. Repeat the link again with the linker option `--no_remove` and examine the image file with `fromelf`.

```
armlink --scatter example.scat --no_remove -o example_scat.axf example.o
fromelf example_scat.axf
```

The output shows that `Section #1` does not have the `SHF_ARM_PURECODE` attribute:

```
...
** Section #1 'ER_MAIN_FOO' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 16 bytes (alignment 4)
    Address: 0x00010000


** Section #2 'ER_REST' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 604 bytes (alignment 4)
    Address: 0x00020000
...
```

The empty RO `.text` section is no longer removed and is placed in the same execution region as `.text.main` and `.text.foo`. Therefore, these sections become read-only.

The same result is obtained when linking with `--keep example.o(.text)` or if there is no `main` or no entry point.

6. To ensure that the sections remain as execute-only, either:
   - Change the scatter file to use the XO attribute selector as follows:

```
LR_XO 0x10000
{
  ER_MAIN_FOO 0x10000
  {
     example.o(+XO)
  }
}

LR_2 0x20000
{
  ER_REST 0x20000
  {
     *(+RO, +ZI)
  }
```

```
  ARM_LIB_STACKHEAP 0x80000 EMPTY -0x1000 {}
}
```

- Explicitly place sections in their execution regions. However, compiling with `-fno-function sections` generates two `.text` sections with different attributes:

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-M3 -mexecute-only -fno-function-
sections -c -o example.o example.c
fromelf example.o
...
** Section #1 '.strtab' (SHT_STRTAB)
    Size   : 107 bytes


** Section #2 '.text' (SHT_PROGBITS) [SHF_ALLOC + SHF_EXECINSTR]
    Size   : 0 bytes (alignment 4)
    Address: 0x00000000


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

In this case, differentiating the sections by name only is not possible. If unused section elimination does not remove the empty `.text` sections, the attribute selectors are required to place the sections in different output sections.

Appendix A
Code Examples

Provides source code examples for Arm Compiler 5 and Arm Compiler 6.

It contains the following sections:

A.1 Example startup code for Arm® Compiler 5 project

This is an example startup code that compiles without errors using Arm Compiler 5.

This code has been modified to demonstrate migration from Arm Compiler 5 to Arm Compiler 6. This code requires other modifications for use in a real application.

`startup_ac5.c:`

```
/*
* Copyright (c) 2009-2017 ARM Limited. All rights reserved.
*
* SPDX-License-Identifier: Apache-2.0
*
* Licensed under the Apache License, Version 2.0 (the License); you may
* not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
* www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an AS IS BASIS, WITHOUT
* WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/


/*----------------------------------------------------------------------------
  Definitions
 *----------------------------------------------------------------------------*/
#define USR_MODE 0x10            // User mode
#define FIQ_MODE 0x11            // Fast Interrupt Request mode
#define IRQ_MODE 0x12            // Interrupt Request mode
#define SVC_MODE 0x13            // Supervisor mode
#define ABT_MODE 0x17            // Abort mode
#define UND_MODE 0x1B            // Undefined Instruction mode
#define SYS_MODE 0x1F            // System mode

/*----------------------------------------------------------------------------
  Internal References
 *----------------------------------------------------------------------------*/
void Vectors       (void) __attribute__ ((section("RESET")));
void Reset_Handler(void);
extern int printf(const char *format, ...);

__declspec(noreturn) void main (void)
{
  __enable_irq();
  printf("Starting main\n");
  while(1);
}
#pragma import (__use_no_semihosting)

/*----------------------------------------------------------------------------
  Exception / Interrupt Handler
 *----------------------------------------------------------------------------*/
void Undef_Handler (void) __attribute__ ((weak, alias("Default_Handler")));
void SVC_Handler   (void) __attribute__ ((weak, alias("Default_Handler")));
void PAbt_Handler  (void) __attribute__ ((weak, alias("Default_Handler")));
void DAbt_Handler  (void) __attribute__ ((weak, alias("Default_Handler")));
void IRQ_Handler   (void) __attribute__ ((weak, alias("Default_Handler")));
void FIQ_Handler   (void) __attribute__ ((weak, alias("Default_Handler")));

/*----------------------------------------------------------------------------
  Exception / Interrupt Vector Table
 *----------------------------------------------------------------------------*/
__asm void Vectors(void) {
  IMPORT Undef_Handler
  IMPORT SVC_Handler
  IMPORT PAbt_Handler
  IMPORT DAbt_Handler
  IMPORT IRQ_Handler
  IMPORT FIQ_Handler
  LDR    PC, =Reset_Handler
  LDR    PC, =Undef_Handler
  LDR    PC, =SVC_Handler
  LDR    PC, =PAbt_Handler
  LDR    PC, =DAbt_Handler
  NOP
  LDR    PC, =IRQ_Handler
```

```
  LDR    PC, =FIQ_Handler
}

/*----------------------------------------------------------------------------
  Reset Handler called on controller reset
 *----------------------------------------------------------------------------*/
__asm void Reset_Handler(void) {

  // Mask interrupts
  CPSID   if

  // Put any cores other than 0 to sleep
  MRC     p15, 0, R0, c0, c0, 5       // Read MPIDR
  ANDS    R0, R0, #3
goToSleep
  WFINE
  BNE     goToSleep

  // Reset SCTLR Settings
  MRC     p15, 0, R0, c1, c0, 0       // Read CP15 System Control register
  BIC     R0, R0, #(0x1 << 12)        // Clear I bit 12 to disable I Cache
  BIC     R0, R0, #(0x1 <<  2)        // Clear C bit  2 to disable D Cache
  BIC     R0, R0, #0x1                // Clear M bit  0 to disable MMU
  BIC     R0, R0, #(0x1 << 11)        // Clear Z bit 11 to disable branch prediction
  BIC     R0, R0, #(0x1 << 13)        // Clear V bit 13 to disable hivecs
  MCR     p15, 0, R0, c1, c0, 0       // Write value back to CP15 System Control register
  ISB

  // Configure ACTLR
  MRC     p15, 0, r0, c1, c0, 1       // Read CP15 Auxiliary Control Register
  ORR     r0, r0, #(1 <<  1)          // Enable L2 prefetch hint (UNK/WI since r4p1)
  MCR     p15, 0, r0, c1, c0, 1       // Write CP15 Auxiliary Control Register

  // Set Vector Base Address Register (VBAR) to point to this application's vector table
  LDR    R0, =Vectors
  MCR    p15, 0, R0, c12, c0, 0

  // Setup Stack for each exceptional mode
  IMPORT |Image$$FIQ_STACK$$ZI$$Limit|
  IMPORT |Image$$IRQ_STACK$$ZI$$Limit|
  IMPORT |Image$$SVC_STACK$$ZI$$Limit|
  IMPORT |Image$$ABT_STACK$$ZI$$Limit|
  IMPORT |Image$$UND_STACK$$ZI$$Limit|
  IMPORT |Image$$ARM_LIB_STACK$$ZI$$Limit|
  CPS    #0x11
  LDR    SP, =|Image$$FIQ_STACK$$ZI$$Limit|
  CPS    #0x12
  LDR    SP, =|Image$$IRQ_STACK$$ZI$$Limit|
  CPS    #0x13
  LDR    SP, =|Image$$SVC_STACK$$ZI$$Limit|
  CPS    #0x17
  LDR    SP, =|Image$$ABT_STACK$$ZI$$Limit|
  CPS    #0x1B
  LDR    SP, =|Image$$UND_STACK$$ZI$$Limit|
  CPS    #0x1F
  LDR    SP, =|Image$$ARM_LIB_STACK$$ZI$$Limit|

  // Call SystemInit
  IMPORT SystemInit
  BL     SystemInit

  // Unmask interrupts
  CPSIE  if


  // Call main
  IMPORT main
  BL     main

}

/*----------------------------------------------------------------------------
  Default Handler for Exceptions / Interrupts
 *----------------------------------------------------------------------------*/
void Default_Handler(void) {
    while(1);
}
```

Related references

B.1 Apache License on page Appx-B-141

A.2 Example startup code for Arm® Compiler 6 project

This is an example startup code that compiles without errors using Arm Compiler 6.

This code has been modified to demonstrate migration from Arm Compiler 5 to Arm Compiler 6. This code requires other modifications for use in a real application.

`startup_ac6.c`:

```
/*
* Copyright (c) 2009-2017 ARM Limited. All rights reserved.
*
* SPDX-License-Identifier: Apache-2.0
*
* Licensed under the Apache License, Version 2.0 (the License); you may
* not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
* www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an AS IS BASIS, WITHOUT
* WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/

/*----------------------------------------------------------------------------
  Definitions
 *----------------------------------------------------------------------------*/
#define USR_MODE 0x10            // User mode
#define FIQ_MODE 0x11            // Fast Interrupt Request mode
#define IRQ_MODE 0x12            // Interrupt Request mode
#define SVC_MODE 0x13            // Supervisor mode
#define ABT_MODE 0x17            // Abort mode
#define UND_MODE 0x1B            // Undefined Instruction mode
#define SYS_MODE 0x1F            // System mode

/*----------------------------------------------------------------------------
  Internal References
 *----------------------------------------------------------------------------*/
void Vectors       (void) __attribute__ ((naked, section("RESET")));
void Reset_Handler (void) __attribute__ ((naked));
extern int printf(const char *format, ...);

__declspec(noreturn) int main (void)
{
  __asm("CPSIE i");
  printf("Starting main\n");
  while(1) __asm volatile("");
}
__asm(".global __use_no_semihosting");

/*----------------------------------------------------------------------------
  Exception / Interrupt Handler
 *----------------------------------------------------------------------------*/
void Undef_Handler (void) __attribute__ ((weak, alias("Default_Handler")));
void SVC_Handler   (void) __attribute__ ((weak, alias("Default_Handler")));
void PAbt_Handler  (void) __attribute__ ((weak, alias("Default_Handler")));
void DAbt_Handler  (void) __attribute__ ((weak, alias("Default_Handler")));
void IRQ_Handler   (void) __attribute__ ((weak, alias("Default_Handler")));
void FIQ_Handler   (void) __attribute__ ((weak, alias("Default_Handler")));

/*----------------------------------------------------------------------------
  Exception / Interrupt Vector Table
 *----------------------------------------------------------------------------*/
void Vectors(void) {
  __asm volatile(
  "LDR    PC, =Reset_Handler                        \n"
  "LDR    PC, =Undef_Handler                        \n"
  "LDR    PC, =SVC_Handler                          \n"
  "LDR    PC, =PAbt_Handler                         \n"
  "LDR    PC, =DAbt_Handler                         \n"
  "NOP                                              \n"
  "LDR    PC, =IRQ_Handler                          \n"
  "LDR    PC, =FIQ_Handler                          \n"
  );
}

/*----------------------------------------------------------------------------
  Reset Handler called on controller reset
```

```
 *----------------------------------------------------------------------------*/
void Reset_Handler(void) {
  __asm volatile(

  // Mask interrupts
  "CPSID   if                                      \n"

  // Put any cores other than 0 to sleep
  "MRC     p15, 0, R0, c0, c0, 5                   \n"  // Read MPIDR
  "ANDS    R0, R0, #3                              \n"
  "goToSleep:                                      \n"
  "WFINE                                           \n"
  "BNE     goToSleep                               \n"

  // Reset SCTLR Settings
  "MRC     p15, 0, R0, c1, c0, 0                   \n"  // Read CP15 System Control register
  "BIC     R0, R0, #(0x1 << 12)                    \n"  // Clear I bit 12 to disable I Cache
  "BIC     R0, R0, #(0x1 <<  2)                    \n"  // Clear C bit  2 to disable D Cache
  "BIC     R0, R0, #0x1                            \n"  // Clear M bit  0 to disable MMU
  "BIC     R0, R0, #(0x1 << 11)                    \n"  // Clear Z bit 11 to disable branch
prediction
  "BIC     R0, R0, #(0x1 << 13)                    \n"  // Clear V bit 13 to disable hivecs
  "MCR     p15, 0, R0, c1, c0, 0                   \n"  // Write value back to CP15 System
Control register
  "ISB                                             \n"

  // Configure ACTLR
  "MRC     p15, 0, r0, c1, c0, 1                   \n"  // Read CP15 Auxiliary Control
Register
  "ORR     r0, r0, #(1 <<  1)                      \n"  // Enable L2 prefetch hint (UNK/WI
since r4p1)
  "MCR     p15, 0, r0, c1, c0, 1                   \n"  // Write CP15 Auxiliary Control
Register

  // Set Vector Base Address Register (VBAR) to point to this application's vector table
  "LDR    R0, =Vectors                             \n"
  "MCR    p15, 0, R0, c12, c0, 0                   \n"

  // Setup Stack for each exceptional mode
  "CPS    #0x11                                    \n"
  "LDR    SP, =Image$$FIQ_STACK$$ZI$$Limit         \n"
  "CPS    #0x12                                    \n"
  "LDR    SP, =Image$$IRQ_STACK$$ZI$$Limit         \n"
  "CPS    #0x13                                    \n"
  "LDR    SP, =Image$$SVC_STACK$$ZI$$Limit         \n"
  "CPS    #0x17                                    \n"
  "LDR    SP, =Image$$ABT_STACK$$ZI$$Limit         \n"
  "CPS    #0x1B                                    \n"
  "LDR    SP, =Image$$UND_STACK$$ZI$$Limit         \n"
  "CPS    #0x1F                                    \n"
  "LDR    SP, =Image$$ARM_LIB_STACK$$ZI$$Limit     \n"

  // Call SystemInit
  "BL     SystemInit                               \n"

  // Unmask interrupts
  "CPSIE  if                                       \n"

  // Call main
  "BL     main                                   \n"
  );
}

/*----------------------------------------------------------------------------
  Default Handler for Exceptions / Interrupts
 *----------------------------------------------------------------------------*/
void Default_Handler(void) {
  while(1);
}
```

Related references

B.1 Apache License on page Appx-B-141

Appendix B
Licenses

Describes the Apache license.

It contains the following section:
- *B.1 Apache License* on page Appx-B-141.

B.1 Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.
2. Grant of Copyright License.

   Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution.
   You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
   a. You must give any other recipients of the Work or Derivative Works a copy of this License; and
   b. You must cause any modified files to carry prominent notices stating that You changed the files; and
   c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
   d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

      You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.
5. Submission of Contributions.

   Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks.

   This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty.

   Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability.

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability.

   While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: HOW TO APPLY THE APACHE LICENSE TO YOUR WORK

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

```
Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

Appendix C
Arm® Compiler Migration and Compatibility Guide Changes

Describes the technical changes that have been made to the Arm Compiler Migration and Compatibility Guide.

It contains the following section:
- *C.1 Changes for the Arm® Compiler Migration and Compatibility Guide* on page Appx-C-145.

C.1 Changes for the Arm® Compiler Migration and Compatibility Guide

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

Table C-1 Changes between 6.16 and 6.15

Change	Topics affected
A note has been added to include a `.balign` directive when defining your own sections with the armclang integrated assembler.	• *5.11 Alignment* on page 5-93.
Added `--diag_style` to the Migration of command-line options section.	• *3.1 Migration of compiler command-line options from Arm® Compiler 5 to Arm® Compiler 6* on page 3-32.
Added note about mixing objects compiled with different C/C++ standards and the change in ABI version.	• *Chapter 6 Changes Between Different Versions of Arm® Compiler 6* on page 6-114.
Added note about the change in ABI version.	• *Chapter 6 Changes Between Different Versions of Arm® Compiler 6* on page 6-114.
Added information to reflect the change in behavior of literal pool merging when using the `PROTECTED` load region attribute.	• *6.1 Summary of changes between Arm® Compiler 6.15 and Arm® Compiler 6.16* on page 6-116.
Added a list of the changes between Arm Compiler 6.15 and Arm Compiler 6.16.	• *6.1 Summary of changes between Arm® Compiler 6.15 and Arm® Compiler 6.16* on page 6-116.

Table C-2 Changes between 6.15 and 6.14

Change	Topics affected
Added comparison for Arm Compiler 5 `--retain` option and Arm Compiler 6 `-O` option.	• *3.1 Migration of compiler command-line options from Arm® Compiler 5 to Arm® Compiler 6* on page 3-32.
Corrected the miscellaneous directives translation table. Mentioned that `.equ` is a synonym for `.set`.	• *5.18 Miscellaneous directives* on page 5-102. • *5.19 Symbol definition directives* on page 5-104.
Improved explanation of when to use the volatile keyword to prevent unwanted removal of inline assembler code when building optimized output.	• *3.4 Inline assembly with Arm® Compiler 6* on page 3-43.
Added details of the new `-Omin` compiler option.	• *2.4 Optimization differences* on page 2-24.
Removed outdated note about using `__ARM_use_no_argv` with `-O0` optimization level in Arm Compiler 6. The `-O0` option now supports `argv` / `argc` optimization.	• *2.4 Optimization differences* on page 2-24.
Updated the entry for the Arm Compiler 5 `--multifile` option.	• *3.1 Migration of compiler command-line options from Arm® Compiler 5 to Arm® Compiler 6* on page 3-32.
Update language extension compatibility section to clarify that the nomerge and notailcall Arm Compiler 5 attributes are not supported in Arm Compiler 6, but that the Community features nomerge and not_tail_called might be considered.	• *4.2 Language extension compatibility: attributes* on page 4-58.

Table C-2 Changes between 6.15 and 6.14 (continued)

Change	Topics affected
Progressive terminology commitment added to Proprietary notices section (all documents).	- Proprietary notices
Added a list of the changes between Arm Compiler 6.14 and Arm Compiler 6.15.	- *6.2 Summary of changes between Arm® Compiler 6.14 and Arm® Compiler 6.15* on page 6-118.