ARM® Compiler

Version 6.6

Migration and Compatibility Guide

ARM DUI0742H

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
H	08 May 2017	Non-Confidential	ARM Compiler v6.6.1 Release

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

http://www.arm.com

Contents

ARM® Compiler Migration and Compatibility Guide

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 9.

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
: Summarizes the support levels, and locales and FlexNet versions supported by the ARM® compilation tools.

Chapter 2 Migrating from armcc to armclang
: Compares ARM Compiler 6 command-line options to older versions of ARM Compiler.

Chapter 3 Compiler Source Code Compatibility
: Provides details of source code compatibility between ARM Compiler 6 and older `armcc` compiler versions.

Chapter 4 Migrating ARM syntax assembly code to GNU syntax
: Describes how to migrate assembly code from the legacy ARM syntax (used by `armasm`) to GNU syntax (used by `armclang`).

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 ARM DUI0742H.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

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

Other information

- *ARM Information Center*.
- *ARM Technical Support Knowledge Articles*.
- *Support and Maintenance*.
- *ARM Glossary*.

Chapter 1
Configuration and Support Information

Summarizes the support levels, and locales 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 and as such, has more functionality than the set of product features described in the documentation. The following definitions clarify the levels of support and guarantees on functionality that are expected from these features.

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

Identification in the documentation

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

Product features

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

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

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

Beta product features

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

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

Alpha product features

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

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

Community features

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

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

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

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

Guidance on use of community features

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

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

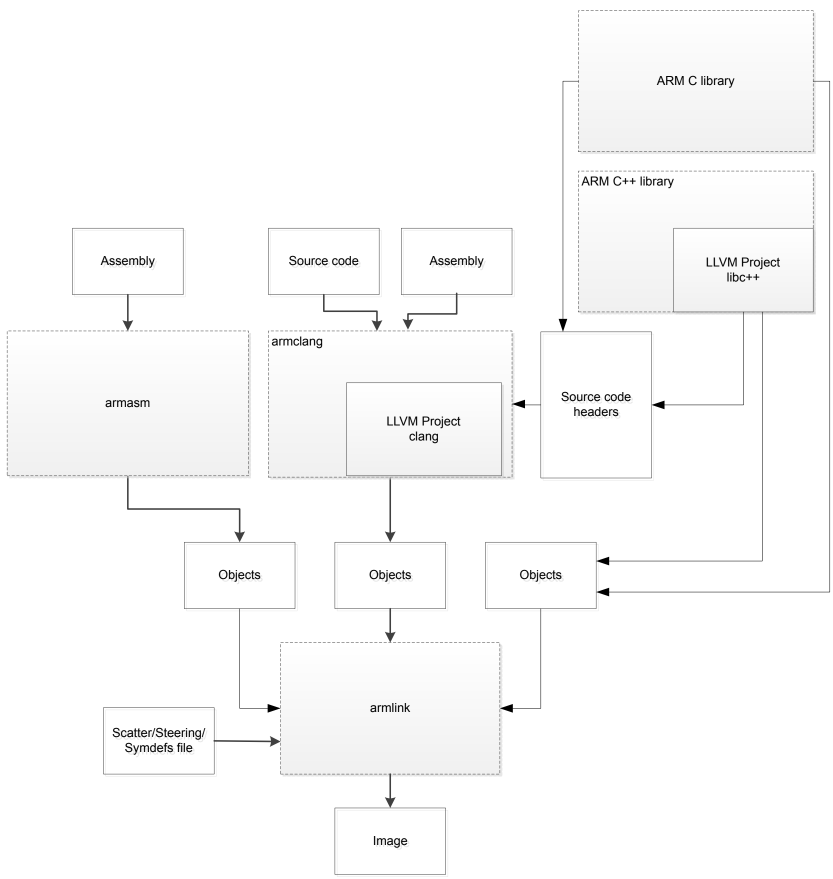

Figure 1-1 Integration boundaries in ARM Compiler 6.

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.

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 1-12.

List of known unsupported features

The following is an incomplete list of unsupported features, and might change over time:
- The Clang option `-stdlib=libstdc++` is not supported.
- C++ static initialization of local variables is not thread-safe when linked against the standard C++ libraries. For thread-safety, you must provide your own implementation of thread-safe functions as described in *Standard C++ library implementation definition*.

  Note

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

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

1.2 Compiler configuration information

Summarizes the locales and 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 6.01 and later	11.12.1.0	11.12.1.0
ARM Compiler 6.00	11.10.1.0	11.10.1.0

Locale support in the compilation tools

ARM Compiler only supports the English locale.

Related information

ARM DS-5 License Management Guide.

Chapter 2
Migrating from armcc to armclang

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

It contains the following sections:

2.1 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 in addition to several ARM-specific options.

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page 1-12.

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

Additional information about command-line options is available:

- The *armclang Reference Guide* provides more detail about a number of command-line options.
- For a full list of Clang command-line options, see the Clang and LLVM documentation.

Table 2-1 Comparison of compiler command-line options in ARM Compiler 6 and ARM Compiler 5

ARM Compiler 5 option	ARM Compiler 6 option	Description
`--allow_fpreg_for_nonfpdata`, `--no_allow_fpreg_for_nonfpdata`	[COMMUNITY] `-mimplicit-float`, `-mno-implicit-float`	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.
`-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.
`--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.

Table 2-1 Comparison of compiler command-line options in ARM Compiler 6 and ARM Compiler 5 (continued)

ARM Compiler 5 option	ARM Compiler 6 option	Description
`--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.
`--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. The default C++ language standard is different between ARM Compiler 5 and ARM Compiler 6.
`--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, AArch32 state.
`--cpu 8-A.64`	`--target=aarch64-arm-none-eabi`	Targets ARMv8-A 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_target`	`-MT`	Changes the target name for the makefile dependency rule.
`--diag_error`	`-Werror`	Turn warnings into errors.
`--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.

Table 2-1 Comparison of compiler command-line options in ARM Compiler 6 and ARM Compiler 5 (continued)

ARM Compiler 5 option	ARM Compiler 6 option	Description
`--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.
`--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.
`--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 2-1 Comparison of compiler command-line options in ARM Compiler 6 and ARM Compiler 5 (continued)

ARM Compiler 5 option	ARM Compiler 6 option	Description
`--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`.
`--no_exceptions`	`-fno-exceptions`	Disables the generation of code needed to support C++ exceptions.
`-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 debug view in ARM Compiler 6, ARM recommends `-O1` rather than `-O0` for best trade-off between image size, performance, and debug.
`-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.

Table 2-1 Comparison of compiler command-line options in ARM Compiler 6 and ARM Compiler 5 (continued)

ARM Compiler 5 option	ARM Compiler 6 option	Description
`--phony_targets`	`-MP`	Emits dummy makefile rules.
`--preinclude`	`-include`	Include the source code of a specified file at the beginning of the compilation.
`--relaxed_ref_def`	`-fcommon`	Places zero-initialized definitions in a common block.
`-S`	`-S`	Outputs the disassembly of the machine code generated by the compiler. The output from this option differs between releases. Older ARM Compiler versions produce 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`.
`--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.
`--use_frame_pointer`, `--no_use_frame_pointer`	`-fno-omit-frame-pointer`, `-fomit-frame-pointer`	Controls whether a register is used for storing stack frame pointers.
`--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 to the language standard.

Table 2-1 Comparison of compiler command-line options in ARM Compiler 6 and ARM Compiler 5 (continued)

ARM Compiler 5 option	ARM Compiler 6 option	Description
`--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

ARM Compiler 6 Command-line Options.
Merging identical constants.
The LLVM Compiler Infrastructure Project.

2.2 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 pre-processing step manually using the `armclang -x assembler-with-cpp` option.
2. Assemble the preprocessed file without using the `--cpreproc` and `--cprepoc_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 >
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.

Specifying a target architecture, processor, and instruction set.

-march armclang option.

-mcpu armclang option.

--target armclang option.

-x armclang option.

Preprocessing assembly code.

2.3 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 2-2 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	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=6-M`	ARMv6-M
`--cpu=6S-M`	`--target=arm-arm-none-eabi -march=armv6s-m`	`--cpu=6S-M`	ARMv6S-M

Table 2-2 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	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 2-3 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 2-3 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 2-3 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 2-3 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 *armclang 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.

Chapter 3
Compiler Source Code Compatibility

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

It contains the following sections:

3.1 Language extension compatibility: keywords

ARM Compiler 6 provides support for some keywords that are supported in ARM Compiler 5.

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page 1-12.

The following table lists some of the commonly used keywords that are supported by ARM Compiler 5 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 is not an exhaustive list of all keywords.

Table 3-1 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))`
`__attribute__((const))`	`__attribute__((const))`
`__forceinline`	`__attribute__((always_inline))`
`__global_reg`	Use inline assembler instructions or equivalent routine.
`__inline(x)`	`__inline__`. The use of this depends on the language mode.
`__int64`	No equivalent. However, you can use `long long`. When you use `long long` in C90 mode, the compiler gives: • a warning. • an error, if you also use `-pedantic-errors`.
`__INTADDR`	[COMMUNITY] None. There is community support for this as a Clang builtin.
`__irq`	`__attribute__((interrupt))`. This is not supported in AArch64.

Table 3-1 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
`__packed` for removing padding within structures.	`__attribute__((packed))`. This provides limited functionality compared to `__packed`: • The `__attribute__((packed))` variable attribute applies to members of a structure or union, but it does not apply to variables that are not members of a struct or union. • `__attribute__((packed))` is not a type qualifier. Taking the address of a packed member can result in unaligned pointers, and in most cases 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` over a structure. 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 provides limited functionality compared to `__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 currently 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`
`__value_in_regs`	`__attribute__((value_in_regs))`
`__weak`	`__attribute__((weak))`
`__writeonly`	No equivalent.

Note

The `__const` keyword was supported by older versions of `armcc`. The equivalent for this keyword in ARM Compiler 5 and ARM Compiler 6 is `__attribute__((const))`.

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 3-2 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 currently no migration. Use a `typedef` instead.
`__packed struct S{...} s;`	`__unaligned struct __attribute__((packed)) S{...} s;` Subsequent declarations of `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

3.6 C and C++ implementation compatibility on page 3-44.
3.2 Language extension compatibility: attributes on page 3-34.
3.3 Language extension compatibility: pragmas on page 3-36.

Related information

__unaligned keyword.

3.2 Language extension compatibility: attributes

ARM Compiler 6 provides support for some function, variable, and type attributes that were supported in ARM Compiler 5. Other attributes are not supported, or have an alternate implementation.

The following attributes are supported by ARM Compiler 5 and ARM Compiler 6. These attributes do not require modification in your code:

- `__attribute__((aligned(x)))`
- `__attribute__((always_inline))`
- `__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__((section("name")))`

  Note

  Section names must be unique. You must not use the same section name for another section or symbol. Only symbols that require the same section type can use the same section name.

- `__attribute__((unused))`
- `__attribute__((used))`

  Note

  In ARM Compiler 6, functions marked with `__attribute__((used))` can still be removed by linker unused section removal. To prevent the linker from removing these sections use the `--keep armlink` option. In ARM Compiler 5, functions marked with `__attribute__((used))` are not removed by the linker.

- `__attribute__((visibility))`
- `__attribute__((weak))`
- `__attribute__((weakref))`

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

Table 3-3 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))`
`__declspec(notshared)`	None. There is no support for BPABI linking models.
`__declspec(thread)`	`__thread`

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 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 3-4 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*
`__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.	ARM Compiler 5 generates an error if the variable has an initializer. Otherwise, it places the zero-initialized variable in a `.bss` section.

Related references

3.6 C and C++ implementation compatibility on page 3-44.
3.1 Language extension compatibility: keywords on page 3-31.
3.3 Language extension compatibility: pragmas on page 3-36.

Related information

armlink User Guide: Placing functions and data in a named section.
armlink User Guide: Placing __at sections at a specific address.

3.3 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 3-5 Pragma language extensions that must be replaced

Pragma supported by older `armcc` compiler versions	Recommended ARM Compiler 6 alternative
`#pragma import (`*`symbol`*`)`	`__asm(".global `*`symbol`*`\n\t");`

`#pragma anon_unions`
 	
`#pragma no_anon_unions`
	In C, anonymous structs and unions are a C11 extension which is enabled by default in armclang. If you specify the `-pedantic` option, the compiler emits warnings about extensions do not match the specified language standard. For example:
	
	`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]`
	
	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 `-pedantic` option, the compiler emits warnings about anonymous structs and classes. For example:
	
	`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]`
	
	Introducing anonymous unions, struct and classes using a `typedef` is a separate extension in `armclang`, which must be enabled using the `-fms-extensions` option.

`#pragma arm`
 	
`#pragma thumb`
	`armclang` does not support switching instruction set in the middle of a file. You can use the command-line options `-marm` and `-mthumb` to specify the instruction set of the whole file.

`#pragma arm section`	`#pragma clang section` In ARM Compiler 5, the section types you can use this pragma with are rodata, rwdata, zidata, and code. In ARM Compiler 6, the equivalent section types are rodata, data, bss, and text respectively. You can also use the `__attribute__((section("name")))` attribute to set the section name of individual functions and variables.

Table 3-5 Pragma language extensions that must be replaced (continued)

Pragma supported by older `armcc` compiler versions	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.

`#pragma pop` `#pragma push`	`armclang` does not support these pragmas. If these are only used to control emission of diagnostics, `#pragma clang diagnostic push` and `#pragma clang diagnostic pop` can be used to achieve the same effect.

Table 3-5 Pragma language extensions that must be replaced (continued)

Pragma supported by older armcc compiler versions	Recommended ARM Compiler 6 alternative
`#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

3.6 C and C++ implementation compatibility on page 3-44.
3.1 Language extension compatibility: keywords on page 3-31.
3.2 Language extension compatibility: attributes on page 3-34.
3.5 Diagnostics for pragma compatibility on page 3-42.

Related information

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

3.4 Language extension compatibility: intrinsics

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

Note

This topic includes descriptions of [COMMUNITY] features. See *Support level definitions* on page 1-12.

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 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. For more information on the ACLE intrinsics, see the *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 3-6 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 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 3-6 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 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. It is an assert if asserts are enabled.	[COMMUNITY] No. However, `__promise` is a community feature.	
`__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.	
`__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	

Table 3-6 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
`__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.	[COMMUNITY] Yes. However, `__swp` is not recommended.	`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	
`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 Cortex-M4 or Cortex-A5 in c99 mode.	No	
`Named register variables`	Allows direct manipulation of a system register as if it were a C variable.	No. To access `FPSCR`, use the `__vfp_status` intrinsic or inline assembly instructions.	

3.5 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 3-7 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 3-7 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:
`-W`*`diag-group`*

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

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

Related references

3.3 Language extension compatibility: pragmas on page 3-36.

3.6 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 3-8 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), -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, unless `--enum_is_int` is specified in C++ mode.	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		
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		

Table 3-8 C and C++ implementation detail differences (continued)

Feature	Older versions of ARM Compiler	ARM Compiler 6
`sizeof(wchar_t)`	2 bytes	4 bytes
`size_t`	Defined as unsigned int, 32-bit.	64-bit on AArch64
`ptrdiff_t`	Defined as signed int, 32-bit.	64-bit on AArch64
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.
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

Related references

3.1 Language extension compatibility: keywords on page 3-31.
3.2 Language extension compatibility: attributes on page 3-34.
3.3 Language extension compatibility: pragmas on page 3-36.
3.7 Compatibility of C++ objects on page 3-46.

3.7 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:

```
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;
}
```

Compiling this with ARM Compiler 5 compiler, `armcc`, and linking with ARM Compiler 6 linker, `armlink`, generates linker errors.

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

This generates the following linker errors:

```
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

armlink User Guide: --stdlib.

Chapter 4
Migrating ARM syntax assembly code to GNU syntax

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

It contains the following sections:

4.1 Overview of differences between ARM and GNU syntax assembly code

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

ARM syntax

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

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

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

        END
```

GNU syntax

```
// Simple GNU syntax example 4.2 Comments on page 4-51//
// Iterate round a loop 10 times, adding 1 to a register each time.

        .section .text,"x"     // 4.6 Sections on page 4-56        .balign
4

main:                          // 4.3 Labels on page 4-52
        MOV      w5,#0x64      // W5 = 100  4.8 Numeric literals on page 4-59
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                   // 4.15 Miscellaneous directives on page 4-66
```

Related references

4.2 Comments on page 4-51.

4.3 Labels on page 4-52.

4.4 Numeric local labels on page 4-53.

4.5 Functions on page 4-55.

4.6 Sections on page 4-56.

4.7 Symbol naming rules on page 4-58.
4.8 Numeric literals on page 4-59.
4.9 Operators on page 4-60.
4.10 Alignment on page 4-61.
4.11 PC-relative addressing on page 4-62.
4.12 Conditional directives on page 4-63.
4.13 Data definition directives on page 4-64.
4.14 Instruction set directives on page 4-65.
4.15 Miscellaneous directives on page 4-66.
4.16 Symbol definition directives on page 4-67.

Related information

About the Unified Assembler Language.

4.2 Comments

A comment identifies text that the assembler ignores.

ARM 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
; As is 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.

armasm User Guide: Syntax of source lines in assembly language.

4.3 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.

ARM 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

4.4 Numeric local labels on page 4-53.

Related information

GNU Binutils - Using as: Labels.

4.4 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.

ARM 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

4.3 Labels on page 4-52.

Related information

GNU Binutils - Using as: Labels.
GNU Binutils - Using as: Local labels.
armasm User Guide: Labels.
armasm User Guide: Numeric local labels.
armasm User Guide: Syntax of numeric local labels.
armasm User Guide: ROUT.

4.5 Functions

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

ARM 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.

armasm User Guide: FUNCTION or PROC.

armasm User Guide: ENDFUNC or ENDP.

4.6 Sections

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

ARM 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 ARM syntax `AREA` attributes map onto GNU syntax `.section` flags. For example, the ARM 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 does not support multiple sections with the same section name. Therefore you must ensure that the different section types have unique names. You must not use the same section name for another section or symbol. If you use the same section name for a different section type, the 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.

armasm User Guide: AREA.

4.7 Symbol naming rules

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

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

- ARM 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 ".".
- ARM 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.
armasm User Guide: Symbol naming rules.

4.8 Numeric literals

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

Implicit shift operations

ARM 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

ARM 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

ARM 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.

armasm User Guide: Syntax of numeric literals.

4.9 Operators

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

The following table shows how to translate ARM syntax operators to GNU syntax operators.

Table 4-1 Operator translation

<table>
<tr><th>ARM syntax operator</th><th>GNU syntax operator</th></tr>
<tr><td>:OR:</td><td>|</td></tr>
<tr><td>:EOR:</td><td>^</td></tr>
<tr><td>:AND:</td><td>&</td></tr>
<tr><td>:NOT:</td><td>~</td></tr>
<tr><td>:SHL:</td><td><<</td></tr>
<tr><td>:SHR:</td><td>>></td></tr>
<tr><td>:LOR:</td><td>||</td></tr>
<tr><td>:LAND:</td><td>&&</td></tr>
<tr><td>:ROL:</td><td>No GNU equivalent</td></tr>
<tr><td>:ROR:</td><td>No GNU equivalent</td></tr>
</table>

Related information

GNU Binutils - Using as: Infix Operators.
armasm User Guide: Unary operators.
armasm User Guide: Shift operators.
armasm User Guide: Addition, subtraction, and logical operators.

4.10 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.

ARM syntax

ARM 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.

ARM 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 25, 1`.

Related information

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

4.11 PC-relative addressing

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

ARM syntax

ARM 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.
armasm User Guide: Register-relative and PC-relative expressions.

4.12 Conditional directives

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

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

Table 4-2 Conditional directive translation

ARM syntax directive	GNU syntax directive
`IF`	`.if`
`IF :DEF:`	`.ifdef`
`IF :LNOT::DEF:`	`.ifndef`
`ELSE`	`.else`
`ELSEIF`	`.elseif`
`ENDIF`	`.endif`

Related information

GNU Binutils - Using as: .if.
GNU Binutils - Using as: .else.
GNU Binutils - Using as: .elseif.
GNU Binutils - Using as: .endif.
armasm User Guide: IF, ELSE, ENDIF, and ELIF.

4.13 Data definition directives

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

The following table shows how to translate ARM 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 4-3 Data definition directives translation

ARM 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.
`DCQ`	`.quad`	Allocate eight-byte blocks of memory, and specify the initial contents.
`SPACE`	`.space`	Allocate a zeroed block of memory.

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.

4.14 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 ARM syntax instruction set directives to GNU syntax directives:

Table 4-4 Instruction set directives translation

ARM 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.
armasm User Guide: ARM or CODE32 directive.
armasm User Guide: CODE16 directive.
armasm User Guide: THUMB directive.

4.15 Miscellaneous directives

Miscellaneous directives perform a range of different functions.

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

Table 4-5 Miscellaneous directives translation

ARM syntax directive	GNU syntax directive	Description
`foo EQU 0x1C`	`.equ foo, 0x1C`	Assigns a value to a symbol. Note the rearrangement of operands.

`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`	Partial support, `armclang` does not fully support `.incbin`.
`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.
`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.
`END`	`.end`	Marks the end of the assembly file.

Related information

GNU Binutils - Using as: .type.
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.
armasm User Guide: ENTRY.
armasm User Guide: END.
armasm User Guide: INFO.
armasm User Guide: EXPORT or GLOBAL.
armlink User Guide: --entry.

4.16 Symbol definition directives

Symbol definition directives declare and set arithmetic, logical, or string variables.

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

Note

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

Table 4-6 Symbol definition directives translation

ARM syntax directive	GNU syntax directive	Description
`LCLA var`	No GNU equivalent	Declare a local arithmetic variable, and initialize its value to 0.
`LCLL var`	No GNU equivalent	Declare a local logical variable, and initialize its value to FALSE.
`LCLS var`	No GNU equivalent	Declare a local string variable, and initialize its value to a null string.
No `armasm` equivalent	`.set var, 0`	Declare a static arithmetic variable, and initialize its value to 0.
No `armasm` equivalent	`.set var, FALSE`	Declare a static logical variable, and initialize its value to FALSE.
No `armasm` equivalent	`.set var, ""`	Declare a static string variable, and initialize its value to a null string.
`GBLA var`	`.global var` `.set var, 0`	Declare a global arithmetic variable, and initialize its value to 0.
`GBLL var`	`.global var` `.set var, FALSE`	Declare a global logical variable, and initialize its value to FALSE.
`GBLS var`	`.global var` `.set var, ""`	Declare a global string variable, and initialize its value to a null string.
`var SETA expr`	`.set var, expr`	Set the value of an arithmetic variable.
`var SETL expr`	`.set var, expr`	Set the value of a logical variable.
`var SETS expr`	`.set var, expr`	Set the value of a string variable.
`foo RN 11`	`foo .req r11`	Define an alias foo for register R11.
`foo QN q5.I32`	`foo .qn q5.i32`	Define an I32-typed alias foo for the quad-precision register Q5.
`foo DN d2.I32`	`foo .dn d2.i32`	Define an I32-typed alias foo for the double-precision register D2.

Related information

GNU Binutils - Using as: ARM Machine Directives.
GNU Binutils - Using as: .global.
GNU Binutils - Using as: .set.