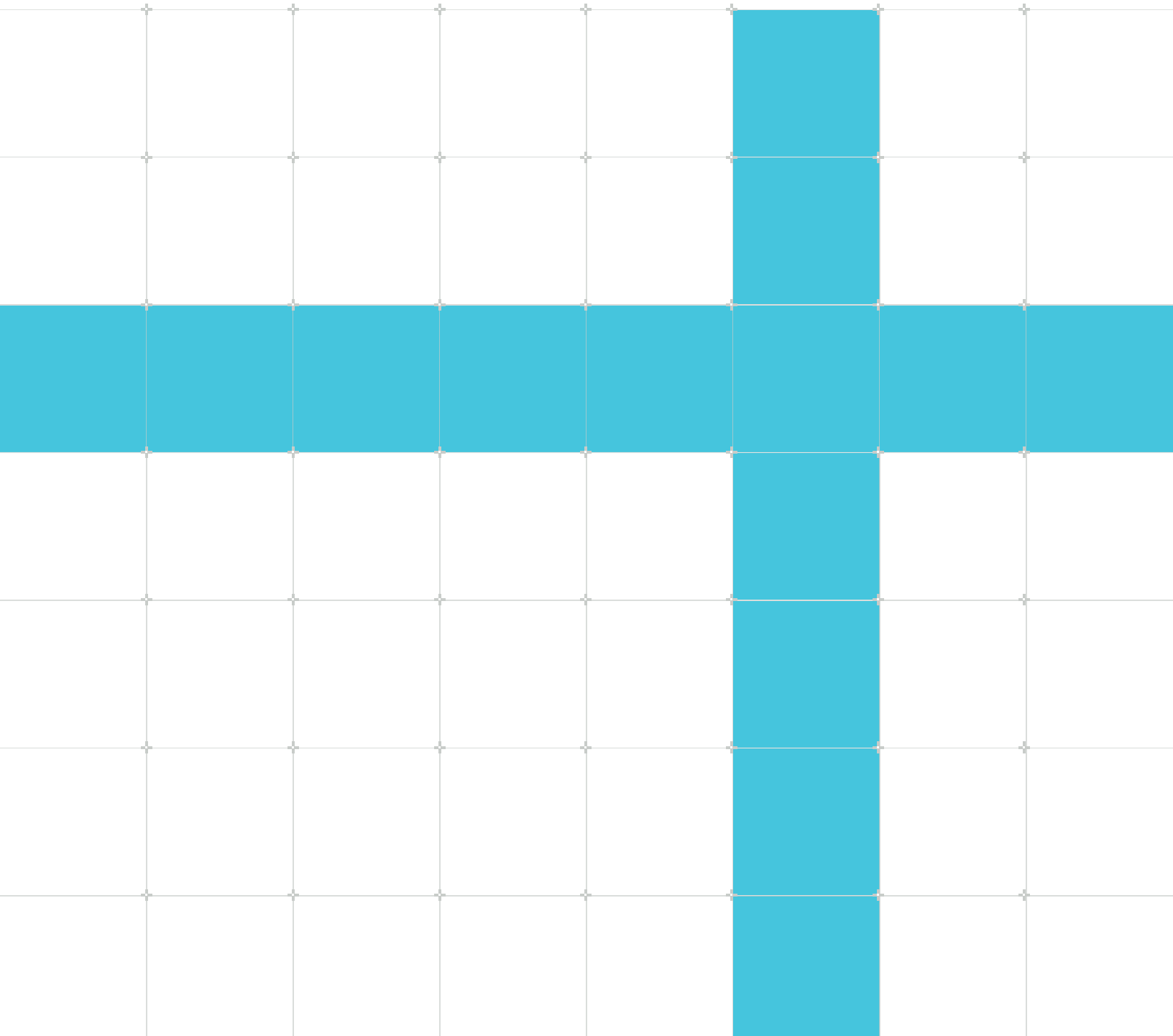
Arm® Compiler

Version 6.6

Migration and Compatibility Guide

Issue
DUI0742_I_en



Arm® Compiler
Migration and Compatibility Guide

Release information

Document history

Issue	Date	Confidentiality	Change
A	14 March 2014	Non-Confidential	Arm Compiler v6.00 Release
B	15 December 2014	Non-Confidential	Arm Compiler v6.01 Release
C	30 June 2015	Non-Confidential	Arm Compiler v6.02 Release
D	18 November 2015	Non-Confidential	Arm Compiler v6.3 Release
E	24 February 2016	Non-Confidential	Arm Compiler v6.4 Release
F	29 June 2016	Non-Confidential	Arm Compiler v6.5 Release
G	4 November 2016	Non-Confidential	Arm Compiler v6.6 Release
H	8 May 2017	Non-Confidential	Arm Compiler v6.6.1 Release
I	29 November 2017	Non-Confidential	Arm Compiler v6.6.2 Release
J	28 August 2019	Non-Confidential	Arm Compiler v6.6.3 Release
K	26 August 2020	Non-Confidential	Arm Compiler v6.6.4 Release
L	31 January 2023	Non-Confidential	Arm Compiler v6.6.5 Release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

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.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

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

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

List of Figures

List of Tables

1. Introduction

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.

1.1 Conventions

The following subsections describe conventions used in Arm documents.

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: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.
Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
Danger	Requirements for the system. Not following these requirements will result in system failure or damage.

Convention	Use
 Note	An important piece of information that needs your attention.
 Tip	A useful tip that might make it easier, better or faster to perform a task.
 Remember	A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm website for other relevant information.

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

2. Configuration and Support Information

Summarizes the support levels and FlexNet versions supported by the Arm compilation tools.

2.1 Support level definitions

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

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

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

Identification in the documentation

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

Product features

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

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

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

Beta product features

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

Beta product features are identified 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 identified 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 more features available in Arm Compiler that are not listed in the documentation. These extra features are known as community features. For information on these community features, see the Clang Compiler User's Manual.

Where community features are referenced in the documentation, they are identified 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 remain functional across update releases, although changes are expected to be unlikely.

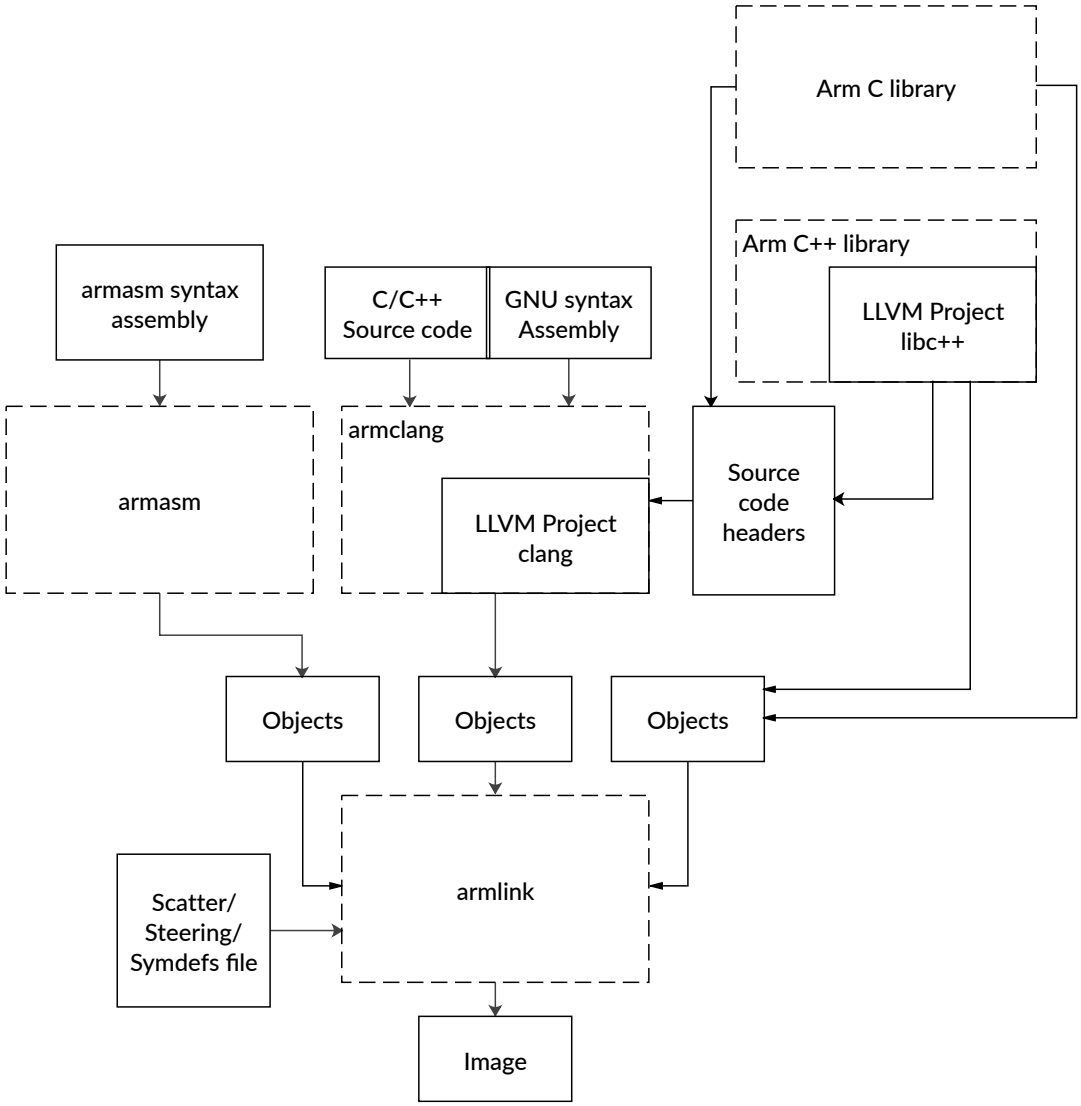
Some community features might become product features in the future, but Arm provides no roadmap for such features. Arm is interested in understanding your use of these features, and welcomes feedback on them. Arm supports customers using these features on a best-effort basis, unless the features are unsupported. Arm accepts defect reports on these features, but does not guarantee that these issues are 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 2-1: Integration boundaries in Arm Compiler for Embedded 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 such features is if the interaction is codified in one of the standards supported by Arm Compiler 6. See Application Binary Interface (ABI). 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 features that have been present for a long time in other toolchains, are likely to be mature. The functionality of new features, such as support for new language features, is likely to be less mature and therefore more likely to have limited functionality.

Deprecated features

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

For information on replacing deprecated features with supported features, see the Arm Compiler documentation and Release Notes. Where appropriate, each Arm Compiler document includes notes for features that are deprecated, and also provides entries in the changes appendix of that document.

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.

List of known unsupported features

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

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

  
  Note

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

- Use of C11 library features is unsupported.
- Any community feature that is exclusively related to non-Arm architectures is not supported.
- Except for Armv6-M, compilation for targets that implement architectures lower than Armv7 is not supported.
- The `long double` data type is not supported for AArch64 state because of limitations in the current Arm C library.
- C complex arithmetic is not supported, because of limitations in the current Arm C library.
- Complex numbers are defined in C++ as a template, `std::complex`. Arm Compiler supports `std::complex` with the `float` and `double` types, but not the `long double` type because of limitations in the current Arm C library.


Note

For C code that uses complex numbers, it is not sufficient to recompile with the C++ compiler to make that code work. How you can use complex numbers depends on whether you are building for Armv8-M architecture-based processors.

- You must take care when mixing translation units that are compiled with and without the [COMMUNITY] `-fsigned-char` option, and that share interfaces or data structures.


Warning

The Arm ABI defines `char` as an unsigned byte, and this is the interpretation used by the C libraries supplied with the Arm compilation tools.

Alternatives to C complex numbers not being supported

If you are building for Armv8-M architecture-based processors, consider using the free and Open Source CMSIS-DSP library that includes a data type and library functions for complex number support in C. For more information about CMSIS-DSP and complex number support see the following sections of the CMSIS documentation:

- Complex Math Functions
- Complex Matrix Multiplication
- Complex FFT Functions

If you are not building for Armv8-M architecture-based processors, consider modifying the affected part of your project to use the C++ standard template library type `std::complex` instead.

2.2 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 2-1: 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

3. Migrating from Arm Compiler 5 to Arm Compiler 6

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

3.1 Migration overview

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

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

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

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

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

Related information

Migrating projects from Arm Compiler 5 to Arm Compiler 6

3.2 Toolchain differences

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

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

Table 3-1: List of compilation tools

Arm Compiler 5	Arm Compiler 6	Function
`armcc`	`armclang`	Compiles C and C++ language source files, including inline assembly.

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

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

Arm Compiler 6 provides `armasm`, which you can use to assemble your existing assembly language source files that are written in `armasm` syntax. Arm recommends that you write new assembly code using the GNU assembly syntax, which you can assemble using the `armclang` integrated assembler. You can also migrate existing assembly language source files from `armasm` syntax to GNU syntax, and then assemble them using the `armclang` integrated assembler. For more information see Migrating from armasm to the armclang Integrated Assembler.

Related information

Migrating projects from Arm Compiler 5 to Arm Compiler 6

3.3 Default differences

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

The following table lists these differences.

Table 3-2: Differences in defaults

Arm Compiler 5	Arm Compiler 6	Notes	Further information
`--apcs=/hardfp` or `--apcs=/softfp`	`-mfloat-abi=softfp`	The default floating-point linkage in Arm Compiler 5 depends on the specified processor. If the processor has floating-point hardware, then Arm Compiler 5 uses hardware floating-point linkage. If the processor does not have floating-point hardware, then Arm Compiler 5 uses software floating-point linkage. In Arm Compiler 6, the default is always software floating-point linkage for AArch32 state. The `-mfloat-abi` option also controls the type of floating-point instructions that the compiler uses. `-mfloat-abi=softfp` uses hardware floating-point instructions. Use `-mfloat-abi=soft` to use software floating-point linkage and software library functions for floating-point operations.	--apcs for Arm Compiler 5. -mfloat-abi for Arm Compiler 6.
`__image.axf`	`a.out`	Default name for the executable image if none of `-o`, `-c`, `-E`, or `-S` are specified on the command line.	-o for Arm Compiler 5. -o for Arm Compiler 6.
`--enum_is_int` is disabled by default	`-fno-short-enums`	`--enum_is_int` is disabled by default in Arm Compiler 5, so the smallest data type that can hold the enumerator values is used. `-fno-short-enums` is the default in Arm Compiler 6, so the size of the enumeration type is at least 32 bits.	--enum_is_int for Arm Compiler 5. -fno-short-enums for Arm Compiler 6.
`-O2`	`-O0`	Arm Compiler 5 uses high optimization (`-O2`) by default. Arm Compiler 6 uses minimum optimization (`-O0`) by default.	-Onum for Arm Compiler 5. -Onum for Arm Compiler 6. Optimization differences.
C++03	C++98	In Arm Compiler 5, the default C++ source language mode is C++03. In Arm Compiler 6, the default source language mode is C++98. You can override the default source language with `-std` in Arm Compiler 6.	--cpp for Arm Compiler 5. -std for Arm Compiler 6.

Arm Compiler 5	Arm Compiler 6	Notes	Further information
C90	C11	In Arm Compiler 5, the default C source language mode C90. In Arm Compiler 6, the default C source language mode C11. You can override the default source language with `-std` in Arm Compiler 6.	--c90 for Arm Compiler 5. -std for Arm Compiler 6.
`--no_exceptions`	`-fexceptions` or `-fno-exceptions`	In Arm Compiler 5, C++ exceptions are disabled by default (`--no_exceptions`). In Arm Compiler 6, C++ exceptions are enabled by default (`-fexceptions`) for C++ sources, or disabled by default (`-fno-exceptions`) for C sources.	--no_exceptions for Arm Compiler 5. -fexceptions for Arm Compiler 6.
`--wchar16`	`-fno-short-wchar`	In Arm Compiler 5, the size of `wchar_t` is 2 bytes by default (`--wchar16`). In Arm Compiler 6, the size of `wchar_t` is 4 bytes by default (`-fno-short-wchar`).	--wchar16 for Arm Compiler 5. -fno-short-wchar for Arm Compiler 6.
`--split_sections` is disabled by default	`-ffunction-sections`	In Arm Compiler 5, functions are not put into separate ELF sections by default (`--split_sections` is disabled). In Arm Compiler 6, each function is put into a separate ELF section by default (`-ffunction-sections`).	--split_sections for Arm Compiler 5. -ffunction-sections for Arm Compiler 6.

3.4 Optimization differences

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

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

Table 3-3: Optimization settings

Description	Arm Compiler 5	Arm Compiler 6
Optimization levels for performance.	• `-Otime -O0` • `-Otime -O1` • `-Otime -O2` • `-Otime -O3` **Note:** The Arm Compiler 5 `-O0` option is more similar to the Arm Compiler 6 `-O1` option than the Arm Compiler 6 `-O0` option.	• `-O0` • `-O1` • `-O2` • `-O3` • `-Ofast` • `-Omax`
Optimization levels for code size.	• `-Ospace -O0` • `-Ospace -O1` • `-Ospace -O2` • `-Ospace -O3` **Note:** The Arm Compiler 5 `-O0` option is more similar to the Arm Compiler 6 `-O1` option than the Arm Compiler 6 `-O0` option.	• `-Os` • `-Oz`
Default	`-Ospace -O2`	`-O0`
Best trade-off between image size, performance, and debug.	`-Ospace -O2`	`-O1`
Highest optimization for performance	`-Otime -O3`	`-Omax`
Highest optimization for code size	`-Ospace -O3`	`-Oz`

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

When using `-Omax`, `armclang` can perform link-time optimizations that were not possible in Arm Compiler 5. These link-time optimizations can expose latent bugs in the final image. Therefore, an image built with Arm Compiler 5 might have a different behavior to the image built with Arm Compiler 6.

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

If the `main()` function has no arguments (no `argc` and `argv`), then Arm Compiler 5 applies a particular optimization at all optimization levels including `-O0`. Arm Compiler 6 applies this optimization only for optimization levels other than `-O0`. When `main()` is compiled with Arm Compiler 6 at any optimization level except -

`O0`, the compiler defines the symbol `__ARM_use_no_argv` if `main()` does not have input arguments. This symbol enables the linker to select an optimized library that does not include code to handle input arguments to `main()`. When `main()` is compiled with Arm Compiler 6 at `-O0`, the compiler does not define the symbol `__ARM_use_no_argv`. Therefore, the linker selects a default library that includes code to handle input arguments to `main()`. This library contains semihosting code. If your `main()` function does not have arguments and you are compiling at `-O0` with Arm Compiler 6, you can select the optimized library by manually defining the symbol `__ARM_use_no_argv` using inline assembly:

```
__asm(".global __ARM_use_no_argv\n\t" "__ARM_use_no_argv:\n\t");
```

Also note that:

- Microlib does not support the symbol `__ARM_use_no_argv`. Only define this symbol when using the standard C library.
- Semihosting code can cause a HardFault on systems that are unable to handle semihosting code. To avoid this HardFault, you must define one or both of:
  - `__use_no_semihosting`
  - `__ARM_use_no_argv`
- If you define `__use_no_semihosting` without `__ARM_use_no_argv`, then the library code to handle `argc` and `argv` requires you to retarget the following functions:
  - `_ttywrch()`
  - `_sys_exit()`
  - `_sys_command_string()`

Related information

-flto armclang option
-O armclang option
Effect of the volatile keyword on compiler optimization
Optimizing across modules with link-time optimization

3.5 Diagnostic messages

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

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

Assignment in condition

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

```
main.cpp:

#include <stdio.h>

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

Compiling this example with Arm Compiler 5 gives the message:

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

Compiling this example with Arm Compiler 6 gives the message:

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

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

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

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

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

Automatic macro expansion

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

Consider the following code:

```
main.cpp:

#include <stdio.h>

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

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

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

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

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

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

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

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

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

For more information, see Diagnostics for pragma compatibility.

When starting the migration from Arm Compiler 5 to Arm Compiler 6, you can expect more diagnostic messages because `armclang` does not recognize some of the pragmas, keywords, and attributes that were specific to `armcc`. When you replace the pragmas, keywords, and attributes from Arm Compiler 5 with their Arm Compiler 6 equivalents, most of these diagnostic messages disappear. If there is no

direct equivalent for Arm Compiler 6, you might require more code changes. For more information, see Compiler Source Code Compatibility.

3.6 Migration example

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


Note

This topic includes descriptions of [COMMUNITY] features. See Support level definitions.

Compiling with Arm Compiler 5

For an example startup code that builds with Arm Compiler 5, see Example startup code for Arm Compiler 5 project.

To compile this example with Arm Compiler 5, enter:

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

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

Compiling with Arm Compiler 6

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

To compile this example with Arm Compiler 6, enter:

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

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

Table 3-4: Command-line changes

Description	Arm Compiler 5	Arm Compiler 6
Tool	`armcc`	`armclang`
Specifying an architecture	`--cpu=7-A`	• `-march=armv7-a` • `--target` is a mandatory option for `armclang`.
Optimization	The default optimization is `-O2`.	The default optimization is `-O0`. To get similar optimizations as the Arm Compiler 5 default, use `-O1`.

Description	Arm Compiler 5	Arm Compiler 6
Source language mode	The default source language mode for `.c` files is `c90`.	The default source language mode for `.c` files is `gnu11` [COMMUNITY]. To compile for `c90` in Arm Compiler 6, use `-std=c90`.

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

```
startup_ac5.c:39:22: error: 'main' must return 'int'
__declspec(noreturn) void main (void)
                     ^~~~
                     int
startup_ac5.c:45:9: error: '#pragma import' is an ARM Compiler 5 extension, and is
 not supported by ARM Compiler 6 [-Warmcc-pragma-import]
#pragma import (__use_no_semihosting)
        ^
startup_ac5.c:60:7: error: expected '(' after 'asm'
__asm void Vectors(void) {
      ^
startup_ac5.c:60:6: error: expected ';' after top-level asm block
__asm void Vectors(void) {
     ^
      ;
startup_ac5.c:61:3: error: use of undeclared identifier 'IMPORT'
  IMPORT Undef_Handler
  ^
startup_ac5.c:80:7: error: expected '(' after 'asm'
__asm void Reset_Handler(void) {
      ^
startup_ac5.c:80:6: error: expected ';' after top-level asm block
__asm void Reset_Handler(void) {
     ^
      ;
startup_ac5.c:83:3: error: use of undeclared identifier 'CPSID'
  CPSID   if
  ^
8 errors generated.
```

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

Modifying the source code for Arm Compiler 6

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

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

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

  With:

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

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

  ```
  __enable_irq();
  ```

With:

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

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

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

With:

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

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

```
while(1);
```

With:

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

4. Migrating from armcc to armclang

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

4.1 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.

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

More information about command-line options is available:

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

Table 4-1: 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`	`-mimplicit-float`, `-mno-implicit-float` [COMMUNITY]	Enables or disables the use of VFP and SIMD registers and data transfer instructions for non-VFP and non-SIMD data.
`--apcs=/nointerwork`	No equivalent.	Disables interworking between A32 and T32 code. Interworking is always enabled in Arm Compiler 6.

`--apcs=/ropi`
 	
`--apcs=/noropi`
	`-fropi`
	
	`-fno-ropi`
		Enables or disables the generation of *Read-Only Position-Independent* (ROPI) code.

`--apcs=/rwpi`
 	
`--apcs=/norwpi`
	`-frwpi`
	
	`-fno-rwpi`
		Enables or disables the generation of *Read/Write Position-Independent* (RWPI) code.

`--arm`	`-marm`	Targets the A32 instruction set. The compiler is permitted to generate both A32 and T32 code, but recognizes that A32 code is preferred.
`--arm_only`	`No equivalent.`	Enforces A32 instructions only. The compiler does not generate T32 instructions.

Arm Compiler 5 option	Arm Compiler 6 option	Description
`--asm`	`-save-temps`	Instructs the compiler to generate intermediate assembly files and 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.
`--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.

Arm Compiler 5 option	Arm Compiler 6 option	Description
`--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.
`--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 option 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.

Arm Compiler 5 option	Arm Compiler 6 option	Description
`--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.
`--library_interface=armcc`	This is the default.	Arm Compiler 6 by default uses the Arm standard C library.
`--library_interface=`*`lib`* Where *`lib`* is one of: • `aeabi_clib` • `aeabi_clib90` • `aeabi_clib99`	`-nostdlib -nostdlibinc -fno-builtin`	Specifies that the compiler output works with any ISO C library compliant with the *Arm Embedded Application Binary Interface* (AEABI).
`--library_interface=`*`lib`* Where *`lib`* is not one of: • `aeabi_clib` • `aeabi_clib90` • `aeabi_clib99` • `armcc`	No equivalent.	Arm Compiler 6 assumes the use of an AEABI-compliant library.

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 option 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.

Arm Compiler 5 option	Arm Compiler 6 option	Description
`-Otime`	This is the default.	Performs optimizations to reduce execution time at the expense of a possible increase in image size. There is no equivalent of the `-Otime` option. Arm Compiler 6 optimizes for execution time by default, unless you specify the `-Os` or `-Oz` options.
`--phony_targets`	`-MP`	Emits dummy makefile rules.
`--preinclude`	`-include`	Include the source code of a specified file at the beginning of the compilation.
`--protect_stack`	`-fstack-protector`, `-fstack-protector-strong`	Enables stack protection on vulnerable functions. See Arm Compiler 5 and Arm Compiler 6 stack protection behavior for more information.
`--protect_stack_all`	`-fstack-protector-all`	Enables stack protection on all functions. See Arm Compiler 5 and Arm Compiler 6 stack protection behavior for more information.
`--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` option `--split_ldm` limits the size of generated `LDM` and `STM` instructions, the `armclang` option `-fno-ldm-stm` 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, `armclang` cannot merge identical constants. Instead, `armlink` does the merge.
`--strict`	`-pedantic-errors`	Generate errors if code violates strict ISO C and ISO C++.

Arm Compiler 5 option	Arm Compiler 6 option	Description
`--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 extra 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.
`--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`.

If the `main()` function has no arguments (no `argc` and `argv`), then Arm Compiler 5 applies a particular optimization at all optimization levels including `-O0`. Arm Compiler 6 applies this optimization only for optimization levels other than `-O0`.

When `main()` is compiled with Arm Compiler 6 at any optimization level except `-O0`, the compiler defines the symbol `__ARM_use_no_argv` if `main()` does not have input arguments. This symbol enables the linker to select an optimized library that does not include code to handle input arguments to `main()`.


Note

When `main()` is compiled with Arm Compiler 6 at `-O0`, the compiler does not define the symbol `__ARM_use_no_argv`. Therefore, the linker selects a default library that includes code to handle input arguments to `main()`. This library contains semihosting code.

If your `main()` function does not have arguments and you are compiling at `-O0` with Arm Compiler 6, you can select the optimized library by manually defining the symbol `__ARM_use_no_argv` using inline assembly:

```
__asm(".global __ARM_use_no_argv\n\t");
```

Also note that:

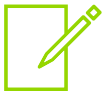
- Semihosting code can cause a HardFault on systems that are unable to handle semihosting code. To avoid this HardFault, you must define one or both of:
  - `__use_no_semihosting`
  - `__ARM_use_no_argv`
- If you define `__use_no_semihosting` without `__ARM_use_no_argv`, then the library code to handle `argc` and `argv` requires you to retarget these functions:
  - `_ttywrch()`
  - `_sys_exit()`
  - `_sys_command_string()`

Related information

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

4.2 Arm Compiler 5 and Arm Compiler 6 stack protection behavior

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



This topic includes descriptions of [COMMUNITY] features. See Support level definitions.

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

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

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


Note

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

Example

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

```
// test.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void *__stack_chk_guard = (void *)0xdeadbeef;

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

static void copy(const char *p) {
  char buf[9];
  strcpy(buf, p);
  printf("Copied: %s\n", buf);
}

int main(void) {
  const char *t = "Hello World!";
  copy(t);
  printf("%s\n", t);
  return 0;
}
```

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

```
armcc -c --cpu=7-A --protect_stack test.c -o test.o
fromelf -c test.o
...
    main
        0x00000010:    e92d407f    .@-.    PUSH     {r0-r6,lr}
        0x00000014:    e28f4064    d@..    ADR      r4,{pc}+0x6c ; 0x80
        0x00000018:    e59f5070    pP..    LDR      r5,[pc,#112] ;
 [__stack_chk_guard = 0x90] = 0
        0x0000001c:    e1a01004    ....    MOV      r1,r4
        0x00000020:    e5950000    ....    LDR      r0,[r5,#0]
        0x00000024:    e58d000c    ....    STR      r0,[sp,#0xc]
        0x00000028:    e1a0000d    ....    MOV      r0,sp
        0x0000002c:    ebfffffe    ....    BL       strcpy
        0x00000030:    e1a0100d    ....    MOV      r1,sp
        0x00000034:    e28f0058    X...    ADR      r0,{pc}+0x60 ; 0x94
        0x00000038:    ebfffffe    ....    BL       __2printf
        0x0000003c:    e59d000c    ....    LDR      r0,[sp,#0xc]
        0x00000040:    e5951000    ....    LDR      r1,[r5,#0]
        0x00000044:    e1500001    ..P.    CMP      r0,r1
```

```
        0x00000048:    1bfffffe    ....    BLNE     __stack_chk_fail ; 0x0
 Section #1
        0x0000004c:    e1a01004    ....    MOV      r1,r4
...
```

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

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

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

Related information

-fstack-protector, -fstack-protector-all, -fstack-protector-strong, -fno-stack-protector
-Rpass

4.3 Command-line options for preprocessing assembly source code

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

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

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

If you have existing source files, which require preprocessing, and that have the lowercase 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 `--cpreproc_opts` options.

Example using armclang -x

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

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

```
armasm --cpu=8-A.64 test_preproc.s
```

Example using armasm --cpreproc_opts

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

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


Note

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

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

4.4 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` options `-mfloat-abi` and `-mfpu` to be compatible with the default floating-point unit in Arm Compiler 5. The tables do not provide an exhaustive list.

Table 4-2: Architecture selection in Arm Compiler 5 and Arm Compiler 6

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

`--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`	Armv7 E-M

Table 4-3: 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

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.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
`--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

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-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
`--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

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

Related information

armclang -mcpu option

5. Compiler Source Code Compatibility

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

5.1 Language extension compatibility: keywords

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


Note

This topic includes descriptions of [COMMUNITY] features. See Support level definitions.

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


Note

This list of keywords is not an exhaustive list.

Table 5-1: 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 keyword depends on the language mode.

Keyword supported by Arm Compiler 5	Recommended Arm Compiler 6 keyword or alternative
`__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`	None. There is [COMMUNITY] support for this keyword as a Clang builtin.
`__irq`	`__attribute__((interrupt))`. This keyword is not supported in AArch64.
`__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 usually the compiler generates a warning. Arm recommends upgrading this warning to an error when migrating code that uses `__packed`. To upgrade the warning to error, use the `armclang` option `-Werror=`*`name`*. The placement of the attribute is different from the placement of `__packed`. If your legacy code contains `typedef __packed struct`, then replace it with: `typedef struct __attribute__((packed))`
`__packed` as a type qualifier for unaligned access.	`__unaligned`. This keyword provides limited functionality compared to the `__packed` type qualifier. The `__unaligned` type qualifier can be used over a structure only when using `typedef` or when declaring a structure variable. This limitation does not apply when using `__packed` in Arm Compiler 5. Therefore, there is 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

Older versions of `armcc` supported the `__const` keyword. 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:

- Removes the padding within structures.
- Qualifies 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 5-2: 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 variables of type `struct S` must use `__unaligned`, for example `__unaligned struct S s2`.
`struct S{__packed int a;}`	`struct S {__attribute__((packed)) __unaligned int a;}`

Related information

__unaligned keyword

5.2 Language extension compatibility: attributes

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


Note

This topic includes descriptions of [COMMUNITY] features. See Support level definitions.

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

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


Note

In Arm Compiler 6, linker unused section removal can still remove functions marked with `__attribute__((used))`. To prevent the linker from removing these sections, you can use either the `--keep=symbol` or the `--no_remove armlink` options. In Arm Compiler 5, the linker does not remove functions marked with `__attribute__((used))`.

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

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

- `__attribute__((nomerge))`

- `__attribute__((notailcall))`

However, because Arm Compiler 6 is built on LLVM technology and preserves the functionality of that technology where possible, you might consider using the following [COMMUNITY] features instead:

- `__attribute__((not_tail_called))`

[COMMUNITY] features are not supported by Arm and are used at your own risk. You are responsible for making sure that any generated code using [COMMUNITY] features is operating correctly. For more information, see Support level definitions.

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

Table 5-3: 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`

__attribute__((always_inline))

Arm Compiler 5 and Arm Compiler 6 support `__attribute__((always_inline))`. However, this attribute might require you to modify your source code.

When using Arm Compiler 5, `__attribute__((always_inline))` forces the inlining of a function with certain exceptions, such as for recursive functions. For more information, see __attribute__((always_inline)) function attribute.

When using Arm Compiler 6, `__attribute__((always_inline))` does not force inlining, but is a hint to the compiler to inline that function. `armclang` still decides whether to inline the function. You can also use the keyword `inline` or `__inline__` (for C90). To determine which functions have been inlined and which ones could not be inlined, you can consider using `-Rpass=inline` and the [COMMUNITY] option `-Rpass-missed=inline`. For more information, see:

- -Rpass.
- __attribute__((always_inline)).
- Inline functions
- Inlining functions.

__attribute__((section("name")))

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

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

When using Arm Compiler 6, you must ensure that variables of different types do not have the same section name.

Arm Compiler 6 supports multiple sections with the same section name only if you use the `.section` assembly directive with a unique ID. For more information, see Section directives.

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

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

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

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

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

Table 5-4: Migrating `__attribute__((at(address)))` and zero-initialized `__attribute__((section("name")))`

Arm Compiler 5 attribute	Arm Compiler 6 attribute	Description
`__attribute__((at(address)))`	`__attribute__((section(".ARM.__at_address")))`	`armlink` in Arm Compiler 6 still supports the placement of sections in the form of `.ARM.__at_address`. **Note:** The Arm Compiler 6 attribute only supports a string to specify the section. To use an arithmetic expression, see Supporting arithmetic expressions in the at(address) attribute in Arm Compiler 6.

Arm Compiler 5 attribute	Arm Compiler 6 attribute	Description
`__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.

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

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

```
#include <stdint.h>

/* Define the variable and place at calculated address directly */
__attribute__((at(0x30000000 + 0x4000))) volatile uint32_t foo = 0x11223344;
```

Arm Compiler 6 does not support arithmetic expressions with `__attribute__((section(".ARM.__at_address")))`. To do the equivalent in Arm Compiler 6, you must use a pointer-based approach as follows:

1. Use pointer arithmetic to define a pointer that points to the required address.
2. The value pointed to by the pointer is not automatically initialized. Instead, you must initialize the value manually within a function.

For example:

```
#include <stdio.h>
#include <stdint.h>

/* 1. Define pointer to point to a calculated address */
volatile uint32_t * const foo_ptr = (volatile uint32_t *) (0x30000000 + 0x4000);

int main(void)
{
    *foo_ptr = 0x11223344; /* 2. Initialize the value pointed to by the pointer */
    printf("%p: %8x\n", foo_ptr, *foo_ptr);  /* Use the pointer as needed */

    return 0;
}
```

Related information

Placing functions and data in a named section
Placing __at sections at a specific address

5.3 Language extension compatibility: pragmas

Arm® Compiler 6 supports 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 5-5: Pragma language extensions that must be replaced

Pragma supported by Arm Compiler 5	Recommended Arm Compiler 6 alternative
`#pragma import (symbol)`	`__asm(".global symbolnt");`

`#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.

Pragma supported by Arm Compiler 5	Recommended Arm Compiler 6 alternative

`#pragma diag_default`
 	
`#pragma diag_suppress`
 	
`#pragma diag_remark`
 	
`#pragma diag_warning`
 	
`#pragma diag_error`
	The following pragmas provide equivalent functionality for `diag_suppress`, `diag_warning`, and `diag_error`:
	• `#pragma clang diagnostic ignored "-Wmultichar"`
	• `#pragma clang diagnostic warning "-Wmultichar"`
	• `#pragma clang diagnostic error "-Wmultichar"`
	
	Note that these pragmas use `armclang` diagnostic groups, which do not have a precise mapping to `armcc` diagnostic tags.
	
	`armclang` has no equivalent to `diag_default` or `diag_remark`. `diag_default` can be replaced by wrapping the change of diagnostic level with `#pragma clang diagnostic push` and `#pragma clang diagnostic pop`, or by manually returning the diagnostic to the default level.
	
	There is an extra 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 supported by Arm Compiler 5	Recommended Arm Compiler 6 alternative

`#pragma pop`
 	
`#pragma push`
	`armclang` does not support these pragmas. Therefore, you cannot push and pop the state of all supported pragmas.
	
	However, you can push and pop the state of the diagnostic pragmas and the state of the pack pragma.
	
	To control the state of the diagnostic pragmas, use `#pragma clang diagnostic push` and `#pragma clang diagnostic pop`.
	
	To control the state of the pack pragma, use `#pragma pack(push)` and `#pragma pack(pop)`.

`#pragma softfp_linkage`	`armclang` does not support this pragma. Instead, use the `__attribute__((pcs("aapcs")))` function attribute to set the calling convention on a per-function basis, or use the `-mfloat-abi=soft` command-line option to set the calling convention on a per-file basis.
`#pragma no_softfp_linkage`	`armclang` does not support this pragma. Instead, use the `__attribute__((pcs("aapcs-vfp")))` function attribute to set the calling convention on a per-function basis, or use the `-mfloat-abi=hard` command-line option to set the calling convention on a per-file basis.

`#pragma unroll[(n)]`
 	
`#pragma unroll_completely`
	`armclang` supports these pragmas.
	
	The default for `#pragma unroll` (that is, with no iteration count specified) differs between `armclang` and `armcc`:
	• With `armclang`, the default is to fully unroll a loop.
	• With `armcc`, the default is `#pragma unroll(4)`.

Related information

#pragma GCC system_header

5.4 Language extension compatibility: intrinsics

Arm® Compiler 6 supports some intrinsics that are supported in Arm Compiler 5.

The following table lists some of the commonly used intrinsics that are supported by Arm Compiler 5 and shows whether Arm Compiler 6 supports them or provides an alternative. If there is no support 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 list is not an exhaustive list of intrinsics.
- The intrinsics provided in `<arm_compat.h>` are only supported for AArch32.

Table 5-6: 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 (Arm®v7 architecture only). Returns previous value of FIQ mask.	Yes	`arm_compat.h`
`__disable_irq`	Disable IRQ interrupts. Returns previous value of IRQ mask.	Yes	`arm_compat.h`
`__dmb`	Inserts a `DMB` instruction or equivalent.	Yes	`arm_acle.h`
`__dsb`	Inserts a `DSB` instruction or equivalent.	Yes	`arm_acle.h`
`__enable_fiq`	Enables fast interrupts.	Yes	`arm_compat.h`
`__enable_irq`	Enables IRQ interrupts.	Yes	`arm_compat.h`
`__fabs`	Inserts a `VABS` or equivalent code sequence.	No. Arm recommends using the standard C library function `fabs()`.	-
`__fabsf`	Single precision version of `__fabs`.	No. Arm recommends using the standard C library function `fabsf()`.	-
`__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 is not optimized away. It also inserts a sequence point, and scheduling barrier for side-effecting function calls.	Yes	`arm_acle.h`

Intrinsic in Arm Compiler 5	Function	Support in Arm Compiler 6	Header file for Arm Compiler 6
`__pld`	Inserts a PLD instruction, if supported.	Yes	`arm_acle.h`
`__pldw`	Inserts a `PLDW` instruction, if supported (Arm®v7 architecture with MP).	No. Arm recommends using `__pldx` described in the ACLE document.	`arm_acle.h`
`__pli`	Inserts a PLI instruction, if supported.	Yes	`arm_acle.h`
`__promise`	Compiler assertion that the expression always has a nonzero value. If asserts are enabled then the promise is checked at runtime by evaluating *`expr`* using `assert(`*`expr`*`)`.	Yes. However, you must `#include <assert.h>` to use `__promise`. `__promise` has the same behavior as `assert()` unless at least one of `NDEBUG` or `__DO_NOT_LINK_PROMISE_WITH_ASSERT` is defined.	`assert.h`
`__qadd`	Inserts a saturating add instruction, if supported.	Yes	`arm_acle.h`
`__qdbl`	Inserts instructions equivalent to `qadd(val,val)`, if supported.	Yes	`arm_acle.h`
`__qsub`	Inserts a saturating subtract, or equivalent routine, if supported.	Yes	`arm_acle.h`
`__rbit`	Inserts a bit reverse instruction.	Yes	`arm_acle.h`
`__rev`	Insert a `REV`, or endian swap instruction.	Yes	`arm_acle.h`
`__return_address`	Returns value of `LR` when returning from current function, without inhibiting optimizations like inlining or tailcalling.	No. Arm recommends using inline assembly instructions.	-
`__ror`	Insert an `ROR` instruction.	Yes	`arm_acle.h`
`__schedule_barrier`	Create a sequence point without effecting memory or inserting `NOP` instructions. Functions with side effects cannot move past the new sequence point.	Yes	`arm_compat.h`
`__semihost`	Inserts an `SVC` or `BKPT` instruction.	Yes	`arm_compat.h`
`__sev`	Insert a `SEV` instruction. Error if the `SEV` instruction is not supported.	Yes	`arm_acle.h`
`__sqrt`	Inserts a `VSQRT` instruction on targets with a VFP coprocessor.	No	-
`__sqrtf`	single-precision version of `__sqrt`.	No	-
`__ssat`	Inserts an `SSAT` instruction. Error if the `SSAT` instruction is not supported.	Yes	`arm_acle.h`
`__strex`	Inserts an appropriately sized Store Exclusive instruction.	No. This intrinsic is deprecated in ACLE 2.0.	-

Intrinsic in Arm Compiler 5	Function	Support in Arm Compiler 6	Header file for Arm Compiler 6
`__strexd`	Inserts a doubleword Store Exclusive instruction.	No. This intrinsic is deprecated in ACLE 2.0.	-
`__strt`	Insert an appropriately sized `STRT` instruction.	No	-
`__swp`	Inserts an appropriately sized `SWP` instruction.	Yes. However, the `SWP` instruction is deprecated, and Arm does not recommend the use of `__swp`.	`arm_acle.h`
`__usat`	Inserts a `USAT` instruction. Error if the `USAT` instruction is not supported.	Yes	`arm_acle.h`
`__wfe`	Inserts a `WFE` instruction. Error if the `WFE` instruction is not supported.	Yes	`arm_acle.h`
`__wfi`	Inserts a `WFI` instruction. Error if the `WFI` instruction is not supported.	Yes	`arm_acle.h`
`__yield`	Inserts a `YIELD` instruction. Error if the `YIELD` instruction is not supported.	Yes	`arm_acle.h`
`ARMv6 SIMD intrinsics`	Inserts an Armv6 SIMD instruction.	No	-
`ETSI intrinsics`	35 intrinsic functions and 2 global variable flags specified in ETSI G729 used for speech encoding. These are provided in the Arm headers in `dspfns.h`.	No	-
`C55x intrinsics`	Emulation of selected TI C55x compiler intrinsics.	No	-
`__vfp_status`	Reads the FPSCR.	Yes	`arm_compat.h`
`FMA intrinsics`	Intrinsics for fused-multiply-add on the Cortex®-M4 or Cortex-A5 processor in c99 mode.	No	-
`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.	-

5.5 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 5-7: 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 Onum`	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`
`#pragma weak symbol`	None	-
`#pragma weak symbol1 = symbol2`	None	-

The following diagnostic groups are also available:

`armcc-pragmas`

Contains all of the diagnostic groups listed in Pragma diagnostics.

`unknown-pragmas`

Contains diagnostics about pragmas which are not known to armclang, and are not in Pragma diagnostics.

`pragmas`

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

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

Suppress a group of diagnostics:

`-Wno-diag-group`

Upgrade a group of diagnostics to warnings:

`-Wdiag-group`

Upgrade a group of diagnostics to errors:

`-Werror=diag-group`

Downgrade a group of diagnostics to warnings:

`-Wno-error=diag-group`

Related information

Language extension compatibility: pragmas on page 51

5.6 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.

Feature	Older versions of Arm Compiler	Arm Compiler 6
Integer operations		
Shifts	`int shifts > 0 && < 127int left_shifts > 31 == 0 int right_shifts > 31 == 0` (for unsigned or positive)`int right_shifts > 31 == -1`(for negative)`long long shifts > 0 && < 63`	Warns when shift amount > width of type.You can use the `-Wshift-count-overflow` option to suppress this warning.
Integer division	Checks that the sign of the remainder matches the sign of the numerator.	The sign of the remainder is not necessarily the same as the sign of the numerator.
Floating-point operations		
Default standard	IEEE 754 standard, rounding to nearest representable value, exceptions disabled by default.	All facilities, operations, and representations guaranteed by the IEEE standard are available in single and double-precision. Modes of operation can be selected dynamically at runtime.This is equivalent to the `--fpmode=ieee_full` option in older versions of Arm Compiler.
`#pragma STDC FP_CONTRACT`	`#pragma STDC FP_CONTRACT`	Might affect code generation.
Unions, enums and structs		

Feature	Older versions of Arm Compiler	Arm Compiler 6
Enum packing	Enums are implemented in the smallest integral type of the correct sign to hold the range of the enum values, except for when compiling in C++ mode with `--enum_is_int`.	By default enums are implemented as `int`, with `long long` used when required.
Allocation of bit fields in containers	Allocation of bit fields in containers.	A container is an object, aligned as the declared type. Its size is sufficient to contain the bit-field, but might be smaller or larger than the bit-field declared type.
Signedness of plain bit fields	Unsigned. Plain bit fields declared without either the `signed` or `unsigned` qualifiers default to `unsigned`. The `--signed_bitfields` option treats plain bit fields as `signed`.	Signed. Plain bit fields declared without either the `signed` or `unsigned` qualifiers default to `signed`. There is no equivalent to either the `--signed_bitfields` or `--no_signed_bitfields` options.
Arrays and pointers		
Casting between integers and pointers	No change of representation	Converting a signed integer to a pointer type with greater bit width sign-extends the integer.Converting an unsigned integer to a pointer type with greater bit width zero-extends the integer.
Misc C		
`sizeof(wchar_t)`	Two bytes	Four bytes
`size_t`	Defined as `unsigned int`, 32-bit.	Defined as `unsigned int` in 32-bit architectures, and *`signtype`* 64-bit in 64-bit architectures.
`ptrdiff_t`	Defined as `signed int`, 32-bit.	Defined as `unsigned int` in 32-bit architectures, and *`signtype`* 64-bit in 64-bit architectures.
Misc C++		
C++ library	Rogue Wave Standard C++ Library	LLVM libc++ Library **Note:** • When the C++ library is used in source code, there is limited compatibility between object code created with Arm Compiler 6 and object code created with Arm Compiler 5. This also applies to indirect use of the C++ library, for example memory allocation or exception handling.
Implicit inclusion	If compilation requires a template definition from a template declared in a header file `xyz.h`, the compiler implicitly includes the file `xyz.cc` or `xyz.CC`.	Not supported.
Alternative template lookup algorithms	When performing referencing context lookups, name lookup matches against names from the instantiation context and 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 information

Language extension compatibility: keywords on page 45
Language extension compatibility: attributes on page 47
Language extension compatibility: pragmas on page 51

5.7 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 use the Rogue Wave header files can be compatible with Arm Compiler 6 objects that use libc++. However, this compatibility is not guaranteed.

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

Compatibility of arrays of objects compiled using Arm Compiler 5

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

```
construct.cpp:
class Foo
```

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

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

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

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

the linker reports:

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

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


Note

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

Related information

--stdlib (armlink)

6. Migrating from armasm to the armclang Integrated Assembler

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

6.1 Overview of differences between armasm and GNU syntax assembly code

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

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

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

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

However, you must change the syntax of your assembly code. These changes include:

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

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

armasm syntax

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

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

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

GNU syntax

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

        .section .text,"x"
        .balign 4
main:
        MOV      w5,#0x64      // W5 = 100
        MOV      w4,#0         // W4 = 0
        B        test_loop     // branch to test_loop
        loop:
        ADD      w5,w5,#1      // Add 1 to W5
        ADD      w4,w4,#1      // Add 1 to W4
        test_loop:
        CMP      w4,#0xa       // if W4 < 10, branch back to loop
        BLT      loop
        .end
```

Related information

Comments on page 64
Labels on page 65
Numeric local labels on page 66
Functions on page 67
Sections on page 68
Symbol naming rules on page 70
Numeric literals on page 70
Operators on page 71
Alignment on page 72
PC-relative addressing on page 73
Conditional directives on page 73
Data definition directives on page 74
Instruction set directives on page 76
Miscellaneous directives on page 76
Symbol definition directives on page 77
About the Unified Assembler Language

6.2 Comments

A comment identifies text that the assembler ignores.

armasm syntax

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

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

For example:

```
; This whole line is a comment
; 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
  ```

6.3 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.

armasm syntax

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

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

GNU syntax

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

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

```
  MOV R0,#16
```

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

6.4 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.

armasm syntax

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

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

Refer to numeric local labels using the following syntax:

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

Where:

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

For example, the following code implements an incrementing loop:

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

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

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

```
routB    ROUT                ; Start of "routB" scope (and therefore end of "routA"
 scope)
```

GNU syntax

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

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

Refer to numeric local labels using the following syntax:

`n{f|b}`

Where:

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

For example, the following code implements an incrementing loop:

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


Note

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

6.5 Functions

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

armasm syntax

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

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

For example:

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

GNU syntax

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

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

GNU syntax assembly code provides the `.func` and `.endfunc` directives. However, `armclang` does support not these directives. `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.

6.6 Sections

Sections are independent, named, indivisible chunks of code or data that the linker can manipulate.

armasm syntax

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

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

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

For example:

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


Note

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

GNU syntax

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

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

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

For example:

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

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

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


Note

When using Arm® Compiler 5, section names do not need to be unique. Therefore, you could use the same section name to create different section types. Arm Compiler 6 supports multiple sections with the same section name only if you specify a unique ID. You must ensure that different section types either:

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

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

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

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

When you assemble the above example code with:

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

The armclang integrated assembler:

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

6.7 Symbol naming rules

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

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

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

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

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

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

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

  GNU syntax symbols do not require double bars:

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

6.8 Numeric literals

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

Implicit shift operations

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

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

Hexadecimal literals

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

For example, the following are equivalent:

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

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

n_base-n-digits format

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

n`_base-`*n*`-digits`

For example:

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

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

For example, you could convert:

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

to:

```
ADD     r1, #13
```

or:

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

6.9 Operators

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

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

Table 6-1: Operator translation

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

6.10 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.

armasm syntax

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

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

GNU syntax

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

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


Note

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

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

6.11 PC-relative addressing

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

armasm syntax

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

For example:

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

GNU syntax

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

```
        ADRP x0, .
```

6.12 Conditional directives

Conditional directives specify conditions that control whether to assemble a sequence of assembly code.

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

Table 6-2: Conditional directive translation

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

armasm syntax directive	GNU syntax directive
ENDIF	.endif

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

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

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

6.13 Data definition directives

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

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


Note

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

Table 6-3: Data definition directives translation

armasm syntax directive	GNU syntax directive	Description
DCB	.byte	Allocate one-byte blocks of memory, and specify the initial contents.
DCW	.hword	Allocate two-byte blocks of memory, and specify the initial contents.
DCD	.word	Allocate four-byte blocks of memory, and specify the initial contents.
DCI	.inst	Allocate a block of memory in the code, and specify the opcode. In A32 code, the block of memory is a four-byte block. In T32 code, the block of memory can be a two-byte or four-byte block. .inst.n allocates a two-byte block and .inst.w allocates a four-byte block.
DCQ	.quad	Allocate eight-byte blocks of memory, and specify the initial contents.

armasm syntax directive	GNU syntax directive	Description

SPACE	.org	Allocate a zeroed block of memory.
		
		The armasm syntax SPACE directive allocates a zeroed block of memory with the specified size. The GNU assembly .org directive zeroes the memory up to the given address. The address must be greater than the address at which the directive is placed.
		
		The following example shows the armasm syntax and GNU syntax methods of creating a 100-byte zeroed block of memory using these directives:
		
		`;  armasm syntax implementation`
		`start_address   SPACE   0x100`
		
		`// GNU syntax implementation`
		`start_address:`
		`.org    start_address + 0x100`
		
		Note:
		If label arithmetic is not required, the GNU assembly .space directive can be used instead of the .org directive. However, Arm recommends using the .org directive wherever possible.

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

armasm syntax	GNU syntax

`Vectors`
`LDR PC, Reset_Addr`
`LDR PC, Undefined_Addr`
`LDR PC, SVC_Addr`
`LDR PC, Prefetch_Addr`
`LDR PC, Abort_Addr`
`B .  ; Reserved vector`
`LDR PC, IRQ_Addr`
`LDR PC, FIQ_Addr`
 	
`Reset_Addr  DCD  Reset_Handler`
`Undefined_Addr  DCD  Undefined_Handler`
`SVC_Addr  DCD  SVC_Handler`
`Prefetch_Addr  DCD  Prefetch_Handler`
`Abort_Addr  DCD  Abort_Handler`
`IRQ_Addr  DCD  IRQ_Handler`
`FIQ_Addr  DCD  FIQ_Handler`
	`Vectors:`
	`ldr pc, Reset_Addr`
	`ldr pc, Undefined_Addr`
	`ldr pc, SVC_Addr`
	`ldr pc, Prefetch_Addr`
	`ldr pc, Abort_Addr`
	`b .  // Reserved vector`
	`ldr pc, IRQ_Addr`
	`ldr pc, FIQ_Addr`
	
	`.balign 4`
	`Reset_Addr:`
	`.word Reset_Handler`
	`Undefined_Addr:`
	`.word Undefined_Handler`
	`SVC_Addr:`
	`.word SVC_Handler`
	`Prefetch_Addr:`
	`.word Prefetch_Handler`
	`Abort_Addr:`
	`.word Abort_Handler`
	`IRQ_Addr:`
	`.word IRQ_Handler`
	`FIQ_Addr:`
	`word FIQ_Handler`

6.14 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 `armasm` syntax instruction set directives to GNU syntax directives:

Table 6-5: Instruction set directives translation

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

6.15 Miscellaneous directives

Miscellaneous directives perform a range of different functions.

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

Table 6-6: Miscellaneous directives translation

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

`EXPORT StartHere`
 	
`GLOBAL StartHere`
	`.global StartHere`
	
	`.type StartHere, @function`
		Declares a symbol that the linker can use. 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`*`.`

armasm syntax directive	GNU syntax directive	Description

`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, &quot;string&quot;`	`.warning &quot;string&quot;`	The `INFO` directive supports diagnostic generation on either pass of the assembly (specified by *n*). The `.warning` directive does not let you specify a particular pass, because the `armclang` integrated assembler only performs one pass.
`ENTRY`	`armlink --entry=`*`location`*	The `ENTRY` directive declares an entry point to a program. `armclang` does not provide an equivalent directive. Use `armlink --entry=`*`location`* to specify the entry point directly to the linker, rather than defining it in the assembly code.
`END`	`.end`	Marks the end of the assembly file.

6.16 Symbol definition directives

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

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


Note

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

Table 6-7: Symbol definition directives translation

armasm 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.

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

`foo DN d2.I32`
 	
`VADD foo, foo, foo`
	`foo .req d2`
	
	`VADD.I32 foo, foo, foo`
		Define an `I32`-typed alias `foo` for the double-precision register `D2`.
		
		When using the `armasm` syntax, you can specify a typed alias for double-precision registers. The example defines an `I32`-typed alias `foo` for the double-precision register `D2`.
		
		When using GNU syntax, you must specify the type on the instruction rather than on the register. The example specifies the `I32` type on the `VADD` instruction.

7. Code Examples

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

7.1 Example startup code for Arm Compiler 5 project

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

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

`startup_ac5.c`:

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


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

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

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

/*----------------------------------------------------------------------------
  Exception / Interrupt Handler
```

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

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

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

  // Mask interrupts
  CPSID   if

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

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

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

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

  // Setup Stack for each exceptional mode
  IMPORT |Image$$FIQ_STACK$$ZI$$Limit|
  IMPORT |Image$$IRQ_STACK$$ZI$$Limit|
  IMPORT |Image$$SVC_STACK$$ZI$$Limit|
  IMPORT |Image$$ABT_STACK$$ZI$$Limit|
  IMPORT |Image$$UND_STACK$$ZI$$Limit|
  IMPORT |Image$$ARM_LIB_STACK$$ZI$$Limit|
```

```
  CPS    #0x11
  LDR    SP, =|Image$$FIQ_STACK$$ZI$$Limit|
  CPS    #0x12
  LDR    SP, =|Image$$IRQ_STACK$$ZI$$Limit|
  CPS    #0x13
  LDR    SP, =|Image$$SVC_STACK$$ZI$$Limit|
  CPS    #0x17
  LDR    SP, =|Image$$ABT_STACK$$ZI$$Limit|
  CPS    #0x1B
  LDR    SP, =|Image$$UND_STACK$$ZI$$Limit|
  CPS    #0x1F
  LDR    SP, =|Image$$ARM_LIB_STACK$$ZI$$Limit|

  // Call SystemInit
  IMPORT SystemInit
  BL     SystemInit

  // Unmask interrupts
  CPSIE  if


  // Call main
  IMPORT main
  BL     main

}

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

7.2 Example startup code for Arm Compiler 6 project

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

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

`startup_ac6.c`:

```
/*

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

/*----------------------------------------------------------------------------
```

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

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

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

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

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

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

  // Mask interrupts
  "CPSID   if                                     \n"

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

  // Reset SCTLR Settings
  "MRC     p15, 0, R0, c1, c0, 0                  \n"  // Read CP15 System Control
 register
```

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

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

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

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

  // Call SystemInit
  "BL      SystemInit                              \n"

  // Unmask interrupts
  "CPSIE   if                                      \n"

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

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

8. Licenses

Describes the Apache license.

8.1 Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/ TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

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

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

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

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

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

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

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

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

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written

communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
    a. You must give any other recipients of the Work or Derivative Works a copy of this License; and
    b. You must cause any modified files to carry prominent notices stating that You changed the files; and
    c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
    d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License. You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any

such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: HOW TO APPLY THE APACHE LICENSE TO YOUR WORK

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

```
Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

      http://www.apache.org/licenses/LICENSE-2.0
```

```
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```