ARM® Compiler

Version 6.00

Migration and Compatibility Guide

ARM

ARM DUI0742A

ARM® Compiler

Migration and Compatibility Guide

Release information

Document History

Issue	Date	Confidentiality	Change
A	14 March 2014	Non-Confidential	ARM Compiler v6.0 Release

Proprietary notice

Confidentiality status

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

Unrestricted Access is an ARM internal classification.

Product status

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

Web address

http://www.arm.com

Contents
ARM® Compiler Migration and Compatibility Guide

List of Tables
ARM® Compiler Migration and Compatibility Guide

Preface

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

This section contains the following subsections:

About this book

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

Using this book

This book is organized into the following chapters:

Chapter 1 Compiler Configuration Information

Summarizes the locales and FlexNet versions supported by the ARM compilation tools.

Chapter 2 Command-line Options Comparison

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

Chapter 3 Compiler Source Code Compatibility

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

Chapter 4 Compiler Migration Support Tools

Describes the set of tools provided by ARM to help with migrating from older compiler versions to ARM Compiler 6.

Typographic conventions

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

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

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

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

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

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

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

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

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

Feedback

Feedback on this product

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

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

Feedback on content

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

- The title.
- The number ARM DUI0742A.
- The page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

——— **Note** ———

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

———————————

Chapter 1
Compiler Configuration Information

Summarizes the locales and FlexNet versions supported by the ARM compilation tools.

It contains the following sections:

1.1 Compiler configuration information

Summarizes the locales and FlexNet versions supported by the ARM compilation tools.

FlexNet versions in the compilation tools

Different versions of ARM® Compiler support different versions of FlexNet.

The FlexNet versions in the compilation tools are:

Table 1-1 FlexNet versions

Compilation tools version	Windows	Linux
ARM Compiler toolchain 6.0	11.10.1.0	11.10.1.0

Locale support in the compilation tools

ARM Compiler only supports the English locale.

Related information

ARM DS-5 License Management Guide.

Chapter 2
Command-line Options Comparison

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

It contains the following sections:

2.1 Comparison of ARM® Compiler 6 compiler command-line options and older versions of ARM® Compiler

Describes the most common ARM Compiler command-line options.

ARM Compiler provides many command-line options, including most Clang command-line options as well as a number of ARM-specific options. Additional information about command-line options is available:

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

Table 2-1 Comparison of ARM Compiler 6 compiler command-line options and older versions of ARM Compiler

Older ARM Compiler option	ARM Compiler 6 option	Description
`-c`	`-c`	Performs the compilation step, but not the link step.
`--c90`	`-xc -std=c90`	Enables the compilation of C90 source code. These are positional arguments and only affect subsequent input files on the command line
`--c99`	`-xc -std=c99`	Enables the compilation of C99 source code. These are positional arguments and only affect subsequent input files on the command line
`--cpp`	`-xc++ -std=c++98`	Enables the compilation of C++ source code. These are positional arguments and only affect subsequent input files on the command line
`--cpu 8-A.32`	`--target=armv8a-arm-none-eabi`	Targets ARMv8, AArch32 state.
`--cpu 8-A.64`	`--target=aarch64-arm-none-eabi`	Targets ARMv8, AArch64 state.
`-D`	`-D`	Defines a preprocessing macro.
`-E`	`-E`	Executes only the preprocessor step.
`-I`	`-I`	Adds the specified directories to the list of places that are searched to find included files.
`--inline`	`-finline-functions`	Enables inlining of functions.
`-L`	`-Xlinker`	Specifies command-line options to pass to the linker when a link step is being performed after compilation.
`-M`	`-M`	Instructs the compiler to produce a list of makefile dependency lines suitable for use by a make utility.
`-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 older compiler versions is `-O2`. The default for ARM Compiler 6 is `-O0`.

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

Older ARM Compiler option	ARM Compiler 6 option	Description
`-Ospace`	`-Oz / -Os`	Performs optimizations to reduce image size at the expense of a possible increase in execution time.
`-Otime`	`(default)`	Performs optimizations to reduce execution time at the expense of a possible increase in image size.
`-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.
`--vectorize`	`-fvectorize`	Enables the generation of Advanced SIMD vector instructions directly from C or C++ code.
`--vsn`	`--version`	Displays version information and license details.

Related information

The LLVM Compiler Infrastructure Project.

2.2 Command-line options for preprocessing assembly source code

The version of `armasm` supplied with ARM Compiler 6 does not support the `--cpreproc` and `--cpreproc_opts` command-line options that were 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 first preprocess the code using `armclang`, then pipe it into `armasm`.

The following example shows the options required to preprocess and assemble the file `example.s`:

```
armclang --target=armv8-arm-eabi-none -E -x assembler-with-cpp example.s | armasm --cpu=8-A.32
-o example.o -
```

Selected command-line options used in this example are:

`-E`
: Specifies that `armclang` only performs preprocessing on the file.

`-x assembler-with-cpp`
: Specifies that `armclang` handles the supplied source file as an assembly source file that requires preprocessing.

`-`
: Specifies that `armasm` reads the assembly source from stdin rather than from a file.

Note

Ensure that you specify compatible architectures in the `armclang --target` option and the `armasm --cpu` option.

Chapter 3
Compiler Source Code Compatibility

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

It contains the following sections:

3.1 Language extension compatibility

ARM Compiler 6 provides support for some language extensions that were supported in older compiler versions. Other language extensions are not supported, or must be replaced with alternatives.

The following table lists some of the commonly used language extensions that are supported by older versions of the compiler but are not supported by ARM Compiler 6. Replace any instances of these language extensions in your code with the recommended alternative.

——— **Note** ———

This is not an exhaustive list of all unsupported language extensions.

———————————

Table 3-1 Language extensions that must be replaced

Language extension supported by older compiler versions	Recommended ARM Compiler 6 alternative
`#pragma import (`*`symbol`*`)`	`asm(" .global `*`symbol`*`\n")`
`__align(x)`	`__attribute__((aligned(x)))`
`__clz`	Use an inline `CLZ` assembly instruction or equivalent routine.
`__const`	`__attribute__((const))`
`__dmb`	Use an inline `DMB` assembly instruction or equivalent CP15 instruction.
`__dsb`	Use an inline `DSB` assembly instruction or equivalent CP15 instruction.
`__forceinline`	`__attribute__((always_inline))`
`__inline`	`__inline__`
`__isb`	Use an inline `ISB` assembly instruction or equivalent CP15 instruction.
`__ldrex`	Use an inline `LDREX` assembly instruction.
`__packed`	`__attribute__((packed, aligned(1)))`
`__pure`	`__attribute__((pure))`
`__rev`	Use an inline `REV` assembly instruction.
`__sev`	Use an inline `SEV` assembly instruction.
`__softfp`	`__attribute__((__pcs__("aapcs")))`
`__strex`	Use an inline `STREX` assembly instruction.
`__weak`	`__attribute__((weak))`
`__wfe`	Use an inline `WFE` assembly instruction.

The following language extensions are supported by older compiler versions and ARM Compiler 6. These language extensions do not require modification in your code:

- `__attribute__((aligned(x)))`
- `__attribute__((always_inline))`
- `__attribute__((const))`
- `__attribute__((deprecated))`
- `__attribute__((nonnull))`
- `__attribute__((noreturn))`
- `__declspec(noreturn)`

- `__declspec(nothrow)`
- `__attribute__((pcs("calling convention")))`
- `__attribute__((pure))`
- `__attribute__((section("name")))`
- `__attribute__((unused))`
- `__attribute__((used))`
- `__attribute__((visibility))`
- `__attribute__((weak))`
- `__attribute__((weakref))`

The following pragmas are the only pragmas supported both by older compiler versions and ARM Compiler 6:

- `#pragma GCC system_header`
- `#pragma once`
- `#pragma pack`
- `#pragma weak`

Related concepts

4.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker on page 4-23.

Related references

3.2 C and C++ implementation compatibility on page 3-18.

3.2 C and C++ implementation compatibility

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

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

Table 3-2 C and C++ implementation detail differences

Feature	Older versions of ARM Compiler	ARM Compiler 6
Integer operations		
Shifts	int shifts > 0 && < 127 int left shifts > 31 == 0 int right shifts > 31 == 0 (for unsigned or +ve), -1 (for -ve) 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.
Unions, enums and structs		
Enum packing	Enums are implemented in the smallest integral type of the correct sign to hold the range of the enum values, unless `--enum_is_int` is specified in C++ mode.	By default enums are implemented as `int`, with `long long` used when required.
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.
Misc C		
`sizeof(wchar_t)`	2 bytes	4 bytes
Misc C++		
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.

Table 3-2 C and C++ implementation detail differences (continued)

Feature	Older versions of ARM Compiler	ARM Compiler 6
Alternative template lookup algorithms	When performing referencing context lookups, name lookup matches against names from the instantiation context as well as from the template definition context.	Not supported.
Exceptions	Off by default, function unwinding on with `--exceptions` by default.	On by default when in C++ mode, but not supported in this release.

Related concepts

4.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker on page 4-23.

Related references

3.1 Language extension compatibility on page 3-16.

Chapter 4
Compiler Migration Support Tools

Describes the set of tools provided by ARM to help with migrating from older compiler versions to ARM Compiler 6.

These tools are as follows:

- The ARM Compiler Source Compatibility Checker.
- The command-line translation wrapper.

It contains the following sections:

4.1 ARM Compiler Source Compatibility Checker command-line syntax

The ARM Compiler Source Compatibility Checker examines C or C++ source code and highlights language extensions that were supported by previous versions of ARM Compilers but are no longer supported by ARM Compiler 6.

The ARM Compiler Source Compatibility Checker is based on Clang and available as a standalone tool, separate from ARM Compiler.

The command for invoking the ARM Compiler Source Compatibility Checker is:

```
compatibility-checker [--version]

compatibility-checker [--no-error] sources [-- compiler-options]
```

where:

sources
: Provides the filenames of one or more text files containing C or C++ source code. By default, the compiler looks for source files in the current directory, but you can also use relative and absolute paths to specify file locations.

compiler-options
: Specifies the `armclang` command-line options used to compile the sources.

  The ARM Compiler Source Compatibility Checker needs to know these compiler options so that it has the same information about your source files as the compiler. For example, if you use the `-I` option with `armclang` to add a directory to the search path, you must provide the same option to the ARM Compiler Source Compatibility Checker so that it searches the same locations for included files.

  Note

  You can only use `armclang` options with the ARM Compiler Source Compatibility Checker. You cannot use `armcc` options.

  Use the `--` separator between the source filenames and these options.

  You can also use a compilation database to specify the compiler options. If you omit `--` *`compiler-options`*, the ARM Compiler Source Compatibility Checker looks for a file named `compile_commands.json` in the directory containing the source files or higher in the directory tree. The `compile_commands.json` file uses the standard Clang JSON compilation database format.

`--version`
: Displays information about the ARM Compiler Source Compatibility Checker, as follows:

```
ARM Compiler Source Compatibility Checker
Software supplied by: ARM Limited
```

`--no-error`
: Specifies that the ARM Compiler Source Compatibility Checker always returns a zero exit code, even if there were errors. Use this option when running the ARM Compiler Source Compatibility Checker from build scripts, where nonzero exit codes can cause problems.

  By default, the ARM Compiler Source Compatibility Checker exits with a nonzero return code if there were any errors, or with zero otherwise.

The ARM Compiler Source Compatibility Checker reports all usage of C or C++ language extensions supported by previous versions of the ARM Compiler in the specified source files. For example:

```
> compatibility-checker /test/myfile.c -- -O2
/test/myfile.c:4:12: warning: armcc extension '__ldrt'
    return __ldrt((const volatile int *)loc);
```

```
           ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

Related concepts

4.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker on page 4-23.

Related tasks

4.3 Specifying compiler options for the ARM Compiler Source Compatibility Checker with a JSON compilation database on page 4-24.

Related references

4.4 JSON compilation database format for the ARM Compiler Source Compatibility Checker on page 4-25.

4.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker

ARM Compiler Source Compatibility Checker checks that your legacy source code does not contain language features that were supported by previous versions of the ARM compiler but are not supported by ARM Compiler 6.

ARM Compiler Source Compatibility Checker checks whether your source code contains a variety of ARM-specific language features, including the following:

- Intrinsics such as `__svc`, `__schedule_barrier`, and `__enable_irq`.
- Attributes such as `__packed`, `__pure`, and `__forceinline`.
- Inline and embedded assembly code.
- Named register variables.
- Macros such as `__OPTIMISE_LEVEL`, `__BIG_ENDIAN`, and `__TARGET_ARCH_ARM`.
- Types such as `__int64`, `long long`, and `__fp16`.
- The `__alignof__` operator.
- Pragmas such as `#pragma diag_suppress`, `#pragma arm`, and `#pragma thumb`.
- GNU builtins such as `__builtin_gamma`, `__builtin_isblank`, and `__builtin__exit`.

Related tasks

4.3 Specifying compiler options for the ARM Compiler Source Compatibility Checker with a JSON compilation database on page 4-24.

Related references

4.4 JSON compilation database format for the ARM Compiler Source Compatibility Checker on page 4-25.

4.1 ARM Compiler Source Compatibility Checker command-line syntax on page 4-21.

3.1 Language extension compatibility on page 3-16.

3.2 C and C++ implementation compatibility on page 3-18.

Related information

The LLVM Compiler Infrastructure Project.

4.3 Specifying compiler options for the ARM Compiler Source Compatibility Checker with a JSON compilation database

The ARM Compiler Source Compatibility Checker needs to know the `armclang` options used to compile the source files. You can use a JSON compilation database to specify these options in a file, rather than the command line, if you prefer.

The ARM Compiler Source Compatibility Checker needs to know these options so that it has the same information about your source files as the compiler. For example, if you use the `-I` option with `armclang` to add a directory to the search path, you must provide the same option to the ARM Compiler Source Compatibility Checker so that it searches the same locations for included files.

Procedure

1. Create a file with the name `compile_commands.json` to contain the compilation database.

   Create this file in the directory containing the source files or higher in the directory tree. The usual location for this file is at the top of the build directory.
2. Create the content of the compilation database to specify the files, directories, and associated compiler commands used by your build system.

   The Clang documentation contains additional information about how to create a compilation database. Search for "JSON Compilation Database Format Specification" on the LLVM Compiler Infrastructure Project web site, `llvm.org`.
3. Run the ARM Compiler Source Compatibility Checker without specifying compiler options on the command line:

   ```
   compatibility-checker hello_world.c
   ```

   Because you did not specify any compiler options, the ARM Compiler Source Compatibility Checker looks for the compilation database `compile_commands.json`, starting in the directory containing the source files (in this example, the current directory) and searching each parent directory in turn until it finds the compilation database.

   Note

   If the ARM Compiler Source Compatibility Checker does not find a compilation database, it exits with an error.

   Note

   You do not have to use a JSON compilation database to specify compiler options. You can specify compiler options on the command line using `compatibility-checker hello_world.c -- compiler-options`.

ARM Compiler Source Compatibility Checker uses the compiler options specified in the compilation database for each source file when checking compatibility.

Related concepts

4.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker on page 4-23.

Related references

4.4 JSON compilation database format for the ARM Compiler Source Compatibility Checker on page 4-25.

4.1 ARM Compiler Source Compatibility Checker command-line syntax on page 4-21.

Related information

The LLVM Compiler Infrastructure Project.

4.4 JSON compilation database format for the ARM Compiler Source Compatibility Checker

A JSON compilation database lets you specify ARM Compiler Source Compatibility Checker options in a file, rather than on the command line.

The Clang documentation contains additional information about how to create a compilation database. Search for "JSON Compilation Database Format Specification" on the LLVM Compiler Infrastructure Project web site, `llvm.org`.

A compilation database specifies an array of command objects, where each command object specifies how a particular source file is compiled. Each command object contains the filename, the working directory of the compile run and the compile command.

For example:

```
[
  { "directory": "/home/user1/build",
    "command": "armclang -O3 hello_world.c",
    "file": "hello_world.c" },
  ...
]
```

Related concepts

4.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker on page 4-23.

Related tasks

4.3 Specifying compiler options for the ARM Compiler Source Compatibility Checker with a JSON compilation database on page 4-24.

Related references

4.1 ARM Compiler Source Compatibility Checker command-line syntax on page 4-21.

4.5 Running the command-line translation wrapper

The command-line translation wrapper lets you run ARM Compiler 6 using older compiler version command-line options.

The command-line translation wrapper is included in the ARM Compiler product installation at *install_dir*/sw/migration/scripts/. It provides a migration path from older versions to ARM Compiler 6.

Note

Add this directory to the $PATH (for Linux users) or PATH system variable (for Windows users).

The command-line translation wrapper converts the specified command-line options into ARM Compiler 6 command-line options, then runs ARM Compiler 6 using these converted command-line options.

The command-line translation wrapper supports only a subset of the older compiler version command-line options, but is customizable so that you can add new conversion mappings for any other command-line options you require. The wrapper is implemented as a Python script and is delivered in source form.

- Run the command-line translation wrapper as follows:

  ```
  armcc opts files
  ```

The compatibility wrapper converts the legacy compiler options you specify, then runs ARM Compiler 6 using these converted command-line options.

For example:

```
armcc --cpu=8-A.32 -Ospace hello_world.c
```

The compatibility wrapper converts this command line to:

```
armclang --target=armv8a-arm-none-eabi -Os hello_world.c
```

Related concepts

4.6 Customizing the command-line translation wrapper on page 4-27.

4.6 Customizing the command-line translation wrapper

The command-line translation wrapper is included in your ARM Compiler product installation at *install_dir*/sw/migration/scripts.

The wrapper is implemented as a Python script and is delivered in source form. It maps a subset of older compiler version command-line options to ARM Compiler 6 command-line options. You can customize the command-line translation wrapper by adding code to perform additional mappings.

The interface is identical to that of the Python `argparse` module. For more information, refer to the `argparse` documentation on the Python programming language web site, `www.python.org`

To convert a single boolean command-line option, use the `add_argument` function. For example, to convert `--foo` to `--bar`, make the following changes in the code:

```
# set up parser with arguments
parser = AC5Parser(prog="armcc")
parser.add_argument('--foo', action='store_true', default=None)
...
if ac5options.foo is not None:
   output_command.append('--bar')
```

To convert a pair of negatable boolean command-line options, that is a pair of complementary options of the form `--option` and `--no_option`, use the `add_negatable_option` function. For example, to convert `--foo` to `--bar` and `--no_foo` to `--pling`, make the following changes in the code:

```
# set up parser with arguments
parser = AC5Parser(prog="armcc")
parser.add_negatable_option('--foo')
...
if ac5options.foo is not None:
  if ac5options.foo:
     output_command.append('--bar')
  else:
     output_command.append('--pling')
```

To convert a pair of complementary boolean command-line options with names that do not follow the format `--option` and `--no_option`, use the `add_complementary_option` function. For example, to convert `--foo` to `--bar` and `--the_opposite_of_foo` to `--pling`, make the following changes in the code:

```
# set up parser with arguments
parser = AC5Parser(prog="armcc")
parser.add_complementary_option('--foo', '--the_opposite_of_foo')
...
if ac5options.foo is not None:
  if ac5options.foo:
     output_command.append('--bar')
  else:
     output_command.append('--pling')
```

The wrapper is documented with comments in the Python script. For full information about how to customize the wrapper, refer to those comments.

Related concepts

4.5 Running the command-line translation wrapper on page 4-26.

Related information

The Python Programming Language web site.