ARM® Compiler

Version 6.01

Migration and Compatibility Guide

ARM

ARM DUI0742B

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

Non-Confidential Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

http://www.arm.com

Contents

ARM® Compiler Migration and Compatibility Guide

List of Tables
ARM® Compiler Migration and Compatibility Guide

Preface

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

It contains the following:

- *About this book* on page 8.

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 `armcc` compiler versions.

Chapter 4 Migrating ARM syntax assembly code to GNU syntax

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

Chapter 5 Compiler Migration Support Tools

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

Glossary

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

See the *ARM Glossary* for more information.

Typographic conventions

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

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

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

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

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

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

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

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

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

Feedback

Feedback on this product

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

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

Feedback on content

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

- The title.
- The number ARM DUI0742B.
- The page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

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

Other information

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

Chapter 1
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 6.01	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

ARM Compiler provides many command-line options, including most Clang command-line options as well as a number of ARM-specific options.

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

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-A, AArch32 state.
`--cpu 8-A.64`	`--target=aarch64-arm-none-eabi`	Targets ARMv8-A, AArch64 state.
`--cpu 7`	`--target=armv7a-arm-none-eabi`	Targets ARMv7-A.
`--cpu=Cortex-A15`	`--target=armv7a-arm-none-eabi -mcpu=cortex-a15`	Targets the Cortex A15 processor.
`-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.
`--licretry`	No equivalent.	There is no equivalent of the `--licretry` option. The ARM Compiler 6 tools automatically retry failed attempts to obtain a license.

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
`-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`.
`-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. 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.
`-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=armv8a-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:

`--target=armv8a-arm-eabi-none`
: Specifies that `armclang` targets ARMv8-A, AArch32 state.

`--cpu=8-A.32`
: Specifies that `armasm` targets ARMv8-A, AArch32 state.

`-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 `armcc` compiler versions.

It contains the following sections:

3.1 Language extension compatibility: keywords

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

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

Note

This is not an exhaustive list of all unsupported keywords.

Table 3-1 Keyword language extensions that must be replaced

Keyword supported by older compiler versions	Recommended ARM Compiler 6 alternative
`__align(x)`	`__attribute__((aligned(x)))`
`__clz`	Use an inline `CLZ` assembly instruction or equivalent routine.
`__const`	`__attribute__((const))`
`__forceinline`	`__attribute__((always_inline))`
`__inline`	`__inline__`
`__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.

Related concepts

5.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker on page 5-47.

Related references

3.5 C and C++ implementation compatibility on page 3-24.

3.2 Language extension compatibility: attributes on page 3-18.

3.3 Language extension compatibility: pragmas on page 3-19.

3.2 Language extension compatibility: attributes

ARM Compiler 6 provides support for some function, variable, and type attributes that were supported in older `armcc` compiler versions. Other attributes are not supported.

The following attributes are supported by older compiler versions and ARM Compiler 6. These attributes 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))`

Related concepts

5.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker on page 5-47.

Related references

3.5 C and C++ implementation compatibility on page 3-24.

3.1 Language extension compatibility: keywords on page 3-17.

3.3 Language extension compatibility: pragmas on page 3-19.

3.3 Language extension compatibility: pragmas

ARM Compiler 6 provides support for some pragmas that were supported in older `armcc` compiler versions. 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 older versions of the compiler but are not supported by ARM Compiler 6. Replace any instances of these pragmas in your code with the recommended alternative.

Table 3-2 Pragma language extensions that must be replaced

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

`#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`	`armclang` does not support setting the sections to be used for code, rodata, rwdata and zidata for the rest of the file. However the `__attribute__((section("name")))` attribute can be used to set the section of individual functions and variables.

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

Table 3-2 Pragma language extensions that must be replaced (continued)

Pragma supported by older armcc compiler versions	Recommended ARM Compiler 6 alternative
`#pragma exceptions_unwind` `#pragma no_exceptions_unwind`	`armclang` does not support C++ exceptions.
`#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. `armclang` stops building the precompiled header at the last preprocessor directive at the top of the main file, and does not provide a mechanism to move the header end point. `armclang` does not provide a mechanism to disable PCH with a `#pragma`. Instead, omit the `-xc-header` or `-xc++-header` options that generate PCH files from the command line. For more information about PCH files, see *Using PCH files to reduce compile time*.
`#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 once`	`armclang` supports this pragma.
`#pragma pack`	`armclang` supports this pragma.
`#pragma pop` `#pragma push`	`armclang` does not support these pragmas. If these are only used to control emission of diagnostics, `#pragma clang diagnostic push` and `#pragma clang diagnostic pop` can be used to achieve the same effect.
`#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)`.
`#pragma weak`	`armclang` supports this pragma.

Related concepts

5.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker on page 5-47.

Related references

3.5 C and C++ implementation compatibility on page 3-24.
3.1 Language extension compatibility: keywords on page 3-17.
3.2 Language extension compatibility: attributes on page 3-18.
3.4 Diagnostics for pragma compatibility on page 3-22.

Related information

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

3.4 Diagnostics for pragma compatibility

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

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

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

Table 3-3 Pragma diagnostics

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

Table 3-3 Pragma diagnostics (continued)

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

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

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

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

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

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

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

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

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

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

Related references

3.3 Language extension compatibility: pragmas on page 3-19.

3.5 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 3-4 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 3-4 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

5.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker on page 5-47.

Related references

3.1 Language extension compatibility: keywords on page 3-17.

3.2 Language extension compatibility: attributes on page 3-18.

3.3 Language extension compatibility: pragmas on page 3-19.

Chapter 4
Migrating ARM syntax assembly code to GNU syntax

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

It contains the following sections:

4.1 Overview of differences between ARM and GNU syntax assembly code

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

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

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

The specific instructions in your assembly code will not change during this migration process (although in some cases, the order of operands may change).

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

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

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

ARM syntax

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

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

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

        END
```

GNU syntax

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

        .section .text,"x"
        .align 2

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
```

4.2 Comments

A comment identifies text that the assembler ignores.

ARM syntax

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

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

For example:

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

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

GNU syntax

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

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

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

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

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

Related information

GNU Binutils - Using as: Comments.

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

4.3 Labels

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

ARM syntax

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

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

GNU syntax

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

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

Related references

4.4 Numeric local labels on page 4-30.

Related information

GNU Binutils - Using as: Labels.

4.4 Numeric local labels

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

ARM syntax

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

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

Refer to numeric local labels using the following syntax:

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

Where:

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

For example, the following code implements an incrementing loop:

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

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

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

GNU syntax

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

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

Refer to numeric local labels using the following syntax:

`n``{f|b}`

Where:

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

For example, the following code implements an incrementing loop:

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

Note

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

Related references

4.3 Labels on page 4-29.

Related information

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

4.5 Functions

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

ARM syntax

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

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

For example:

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

GNU syntax

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

```
  .type myproc, @function
myproc:
  ; Procedure body
```

Note

GNU syntax assembly code provides the `.func` and `.endfunc` directives. However, these are not supported by `armclang`.

Note

Exported functions must be typed to link properly.

Related information

GNU Binutils - Using as: .type.
armasm Reference Guide: FUNCTION or PROC.
armasm Reference Guide: ENDFUNC or ENDP.

4.6 Sections

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

ARM syntax

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

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

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

For example:

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

Note

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

GNU syntax

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

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

- `x` specifies that the section is executable.
- `w` specifies that the section is writable.

For example:

```
.section  mysection,"x"
```

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

```
.section  mysection,"x"
.align 3
```

Related information

GNU Binutils - Using as: .section.
GNU Binutils - Using as: .align.
armasm Reference Guide: AREA.

4.7 Symbol naming rules

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

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

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

Symbol naming rules which differ between ARM syntax and GNU syntax include:

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

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

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

  GNU syntax symbols do not require double bars:

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

Related information

GNU Binutils - Using as: Symbol Names.
armasm User Guide: Symbol naming rules.

4.8 Numeric literals

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

Hexadecimal literals

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

For example, the following are equivalent:

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

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

n_base-n-digits format

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

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

For example:

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

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

For example, you could convert:

```
ADD     r1, 2_1101
```

to:

```
ADD     r1, #13
```

or:

```
ADD     r1, #0xB
```

Related information

GNU Binutils - Using as: Integers.
armasm User Guide: Numeric literals.

4.9 Operators

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

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

Table 4-1 Operator translation

ARM syntax operator	GNU syntax operator
:OR:	|
:AND:	&
:NOT:	~
:SHL:	<<
:SHR:	>>

Related information

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

4.10 Alignment

Data and code must be aligned to appropriate boundaries.

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

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

ARM syntax

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

ARM syntax assembly also provides the `PRESERVE8` directive. The `PRESERVE8` directive specifies that the current file preserves eight-byte alignment of the stack. GNU syntax assembly does not provide an equivalent to this directive.

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.

Related information

GNU Binutils - Using as: .align.
GNU Binutils - Using as: .balign.
armasm Reference Guide: REQUIRE8 and PRESERVE8.
armasm Reference Guide: ALIGN.

4.11 PC-relative addressing

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

ARM syntax

ARM syntax assembly provides the symbol `{pc}` to let you use the address of the program counter.

For example:

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

GNU syntax

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

```
ADRP x0, .
```

Related information

GNU Binutils - Using as: The Special Dot Symbol.

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

4.12 Conditional directives

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

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

Table 4-2 Conditional directive translation

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

Related information

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

4.13 Data definition directives

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

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

Note

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

Table 4-3 Data definition directives translation

ARM syntax directive	GNU syntax directive	Description
`DCB`	`.byte`	Allocate one-byte blocks of memory, and specify the initial contents.
`DCW`	`.hword`	Allocate two-byte blocks of memory, and specify the initial contents.
`DCD`	`.word`	Allocate four-byte blocks of memory, and specify the initial contents.
`DCQ`	`.quad`	Allocate eight-byte blocks of memory, and specify the initial contents.
`SPACE`	`.space`	Allocate a zeroed block of memory.

Related information

GNU Binutils - Using as: .byte.
GNU Binutils - Using as: .word.
GNU Binutils - Using as: .hword.
GNU Binutils - Using as: .quad.
GNU Binutils - Using as: .space.
armasm Reference Guide: Data definition directives.

4.14 Instruction set directives

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

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

Table 4-4 Instruction set directives translation

ARM syntax directive	GNU syntax directive	Description
`ARM` or `CODE32`	`.arm` or `.code32`	Interpret subsequent instructions as A32 instructions.
`THUMB` or `CODE16`	`.thumb` or `.code16`	Interpret subsequent instructions as T32 instructions.

Related information

GNU Binutils - Using as: ARM Machine Directives.
armasm Reference Guide: ARM, THUMB, CODE16 and CODE32.

4.15 Miscellaneous directives

Miscellaneous directives perform a range of different functions.

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

Table 4-5 Miscellaneous directives translation

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

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

`GET file` `INCLUDE file`	`.include file`	Includes a file within the file being assembled.
`IMPORT foo`	`.global foo`	Provides the assembler with a name that is not defined in the current assembly.
`INCBIN`	`.incbin`	Partial support, `armclang` does not fully support `.incbin`.
`INFO n, "string"`	`.print "string"`	The `INFO` directive supports diagnostic generation on either pass of the assembly (specified by *n*). The `.print` directive does not let you specify a particular pass.
`ENTRY`	`armlink --entry=Location`	The `ENTRY` directive declares an entry point to a program. `armclang` does not provide an equivalent directive. Use `armlink --entry=Location` to specify the entry point directly to the linker, rather than defining it in the assembly code.
`END`	`.end`	Marks the end of the assembly file.

Related information

GNU Binutils - Using as: .type.
GNU Binutils - Using as: .print.
GNU Binutils - Using as: .equ.
GNU Binutils - Using as: .global.
GNU Binutils - Using as: .include.
GNU Binutils - Using as: .incbin.
armasm Reference Guide: ENTRY.
armasm Reference Guide: END.
armasm Reference Guide: Miscellaneous directives.
armasm Reference Guide: INFO.
armasm Reference Guide: EXPORT or GLOBAL.
armlink Reference Guide: --entry.

4.16 Symbol definition directives

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

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

Note

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

Table 4-6 Symbol definition directives translation

ARM syntax directive	GNU syntax directive	Description
`LCLA var`	`.set var, 0`	Declare a local arithmetic variable, and initialize its value to 0.
`LCLL var`	`.set var, FALSE`	Declare a local logical variable, and initialize its value to FALSE.
`LCLS var`	`.set var, ""`	Declare a local 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 .reg r11`	Define an alias foo for register R11.
`foo QN q5.I32`	`foo .qn q5.i32`	Define an I32-typed alias foo for the quad-precision register Q5.
`foo DN d2.I32`	`foo .dn d2.i32`	Define an I32-typed alias foo for the double-precision register D2.

Related information

GNU Binutils - Using as: ARM Machine Directives.
GNU Binutils - Using as: .global.
GNU Binutils - Using as: .set.
armasm Reference Guide: Symbol definition directives.

Chapter 5
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:

5.1 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

5.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker on page 5-47.

Related tasks

5.3 Specifying compiler options for the ARM Compiler Source Compatibility Checker with a JSON compilation database on page 5-48.

Related references

5.4 JSON compilation database format for the ARM Compiler Source Compatibility Checker on page 5-50.

5.2 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

5.3 Specifying compiler options for the ARM Compiler Source Compatibility Checker with a JSON compilation database on page 5-48.

Related references

5.4 JSON compilation database format for the ARM Compiler Source Compatibility Checker on page 5-50.

5.1 ARM Compiler Source Compatibility Checker command-line syntax on page 5-45.

3.5 C and C++ implementation compatibility on page 3-24.

3.1 Language extension compatibility: keywords on page 3-17.

3.2 Language extension compatibility: attributes on page 3-18.

3.3 Language extension compatibility: pragmas on page 3-19.

Related information

The LLVM Compiler Infrastructure Project.

5.3 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, *http://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`.

Postrequisites

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

Related concepts

5.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker on page 5-47.

Related references

5.4 JSON compilation database format for the ARM Compiler Source Compatibility Checker on page 5-50.

5.1 ARM Compiler Source Compatibility Checker command-line syntax on page 5-45.

Related information

The LLVM Compiler Infrastructure Project.

5.4 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, `http://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

5.2 Compatibility checks performed by ARM Compiler Source Compatibility Checker on page 5-47.

Related tasks

5.3 Specifying compiler options for the ARM Compiler Source Compatibility Checker with a JSON compilation database on page 5-48.

Related references

5.1 ARM Compiler Source Compatibility Checker command-line syntax on page 5-45.

5.5 Running the command-line translation wrapper

The command-line translation wrapper lets you run ARM Compiler 6 using `armcc` 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 `armcc` 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`* `[-v | --verbose]` *`files`*

Where:

`opts`
: Specifies the `armcc` command-line options you want to translate.

`-v` or `--verbose`
: Displays verbose output from the command-line translation wrapper.

`files`
: Provides the filenames of one or more text files containing source code.

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

5.6 Customizing the command-line translation wrapper on page 5-52.

5.6 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

5.5 Running the command-line translation wrapper on page 5-51.

Related information

The Python Programming Language web site.