ARM Compiler

Version 6.02

armclang Reference Guide

ARM

ARM DUI0774C

ARM Compiler

armclang Reference 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

Non-Confidential Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Additional Notices

Some material in this document is based on IEEE 754-1985 IEEE Standard for Binary Floating-Point Arithmetic. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner.

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 armclang Reference Guide

List of Figures
ARM Compiler armclang Reference Guide

List of Tables
ARM Compiler armclang Reference Guide

Preface

This preface introduces the *ARM Compiler armclang Reference Guide*.

It contains the following:

About this book

The ARM® Compiler armclang Reference Guide provides user information for the ARM compiler, `armclang`. `armclang` is an optimizing C and C++ compiler that compiles Standard C and Standard C++ source code into machine code for ARM architecture-based processors.

Using this book

This book is organized into the following chapters:

Chapter 1 Compiler Command-line Options
Summarizes the most common options used with `armclang`.

Chapter 2 Compiler-specific Keywords and Operators
Summarizes the compiler-specific keywords and operators that are extensions to the C and C++ Standards.

Chapter 3 Compiler-specific Function, Variable, and Type Attributes
Summarizes the compiler-specific function, variable, and type attributes that are extensions to the C and C++ Standards.

Chapter 4 Compiler-specific Pragmas
Summarizes the ARM compiler-specific pragmas that are extensions to the C and C++ Standards.

Chapter 5 Other Compiler-specific Features
Summarizes compiler-specific features that are extensions to the C and C++ Standards, such as predefined macros.

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 DUI0774C.
- 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 Command-line Options

Summarizes the most common options used with `armclang`.

`armclang` provides many command-line options, including most Clang command-line options in addition to a number of ARM-specific options. Additional information about command-line options is available in the Clang and LLVM documentation on the LLVM Compiler Infrastructure Project web site, *http://llvm.org*.

Note

Be aware of the following:

- Generated code might be different between two ARM® Compiler releases.
- For a feature release, there might be significant code generation differences.

Note

The command-line option descriptions and related information in the individual ARM Compiler tools documents describe all the features that are specific to, and supported by, ARM Compiler. Any features specific to ARM Compiler that are not documented are not supported and are used at your own risk. Although open-source `clang` LLVM features are available, they are not supported by ARM and are used at your own risk. You are responsible for making sure that any generated code using unsupported features is operating correctly.

It contains the following sections:

1.1 -c

Instructs the compiler to perform the compilation step, but not the link step.

Usage

ARM recommends using the `-c` option in projects with more than one source file.

The compiler creates one object file for each source file, with a `.o` file extension replacing the file extension on the input source file. For example, the following creates object files `test1.o`, `test2.o`, and `test3.o`:

```
armclang --target=aarch64-arm-none-eabi -c test1.c test2.c test3.c
```

Note

If you specify multiple source files with the `-c` option, the `-o` option results in an error. For example:

```
armclang --target=aarch64-arm-none-eabi -c test1.c test2.c -o test.o
armclang: error: cannot specify -o when generating multiple output files
```

1.2 -D

Defines the macro *name*.

Syntax

`-Dname[(parm-list)][=def]`

Where:

name
: Is the name of the macro to be defined.

parm-list
: Is an optional list of comma-separated macro parameters. By appending a macro parameter list to the macro name, you can define function-style macros.
The parameter list must be enclosed in parentheses. When specifying multiple parameters, do not include spaces between commas and parameter names in the list.

  Note

  Parentheses might require escaping on UNIX systems.

=def
: Is an optional macro definition.

  If *=def* is omitted, the compiler defines *name* as the value 1.

  To include characters recognized as tokens on the command line, enclose the macro definition in double quotes.

Usage

Specifying `-Dname` has the same effect as placing the text `#define name` at the head of each source file.

Example

Specifying this option:

```
-DMAX(X,Y)="((X > Y) ? X : Y)"
```

is equivalent to defining the macro:

```
#define MAX(X, Y) ((X > Y) ? X : Y)
```

at the head of each source file.

1.3 -E

Executes the preprocessor step only.

By default, output from the preprocessor is sent to the standard output stream and can be redirected to a file using standard UNIX and MS-DOS notation.

You can use the `-o` option to specify a file for the preprocessed output.

By default, comments are stripped from the output. Use the `-C` option to keep comments in the preprocessed output.

To generate interleaved macro definitions and preprocessor output, use `-E -dD`.

Example

```
armclang --target=aarch64-arm-none-eabi -E -dD source.c > raw.c
```

1.4 -e

Specifies the unique initial entry point of the image.

`armclang` translates this option to `--entry` and passes it to `armlink`.

See the *ARM® Compiler toolchain Linker Reference* for information about the `--entry` linker options.

Related information

ARM Compiler toolchain Linker Reference.

1.5 -fbare-metal-pie

Generates position independent code.

This option causes the compiler to invoke `armlink` with the `--bare_metal_pie` option when performing the link step.

——— **Note** ———

Not supported for AArch64 state.

———————————

Related information

Bare-metal Position Independent Executables.
--fpic armlink option.
--pie armlink option.
--bare_metal_pie armlink option.
--ref_pre_init armlink option.

1.6 -fcommon, -fno-common

Generates common zero-initialized definitions for tentative definitions.

Tentative definitions are declarations of variables with no storage class and no initializer.

The `-fno-common` option generates individual zero-initialized definitions for tentative definitions. These zero-initialized definitions are placed in a ZI section in the generated object. Multiple definitions in different files cause a `L6200E: Symbol multiply defined` linker error because the individual definitions clash with each other.

The `-fcommon` option places the zero-initialized definitions in a common block. This common definition is not associated with any particular section or object, so multiple definitions resolve to a single definition at link time.

Default

The default is `-fcommon`.

1.7 -ffast-math

Setting this option tells the compiler to perform more aggressive floating-point optimizations.

It results in behavior that is not fully compliant with the ISO C or C++ standard. However, numerically robust floating-point programs are expected to behave correctly.

1.8 @file

Reads a list of compiler options from a file.

Syntax

@file

Where *file* is the name of a file containing armclang options to include on the command line.

Usage

The options in the specified file are inserted in place of the *@file* option.

Use whitespace or new lines to separate options in the file. Enclose strings in single or double quotes to treat them as a single word.

You can specify multiple *@file* options on the command line to include options from multiple files. Files can contain more *@file* options.

If any *@file* option specifies a non-existent file or circular dependency, armclang exits with an error.

Example

Consider a file options.txt with the following content:

```
"-I../my libs/"
--target=aarch64-arm-none-eabi -mcpu=cortex-a57
```

Compile a source file main.c with the following command line:

```
armclang @options.txt main.c
```

This command is equivalent to the following:

```
armclang -I"../my libs/" --target=aarch64-arm-none-eabi -mcpu=cortex-a57 main.c
```

1.9 -fno-inline-functions

Disabling the inlining of functions can help to improve the debug experience.

When the option `-fno-inline-functions` is selected at optimization levels `-O2` and higher, the compiler does not attempt to automatically inline functions.

Related concepts

5.2 Inline functions on page 5-117.

Related references

1.32 -O on page 1-47.

1.10 -flto

Enables link time optimization, outputting bitcode files for link time optimization rather than ELF object files.

The primary use for bitcode files is for link time optimization. See *Optimizing across modules with link time optimization* in the *Software Development Guide* for more information about link time optimization.

Note

ARM Compiler does not support link time optimization on 32-bit Red Hat Enterprise Linux platforms.

Usage

The compiler creates one bitcode file for each source file, with a `.o` file extension replacing the file extension on the input source file.

The `-flto` option passes the `--lto` option to `armlink` to enable link time optimization, unless the `-c` option is specified.

Related references

1.1 -c on page 1-14.

Related information

Optimizing across modules with link time optimization.

--lto armlink option.

1.11 -fno-exceptions

Suppresses the generation of code needed to support C++ exceptions.

Usage

The `-fno-exceptions` option can be used with the C++ Standard Library libc++, with caution:

- ARM Compiler 6 does not contain variants of libc++ that are built with `-fno-exceptions`. Therefore, exceptions might be thrown from pre-built libc++ objects.
- The parts of libc++ that are implemented in header files might be compiled with `-fno-exceptions`. When `-fno-exceptions` is used, libc++ calls the assert macro instead of throwing an exception.

  Note

  If the macro `NDEBUG` is defined these assert macros are removed.

Use of try, catch, or throw results in an error message. If a C++ exception from another object propagates into code built with `-fno-exceptions`, then the program terminates.

Rogue Wave C++ Libraries

`armclang` and Rogue Wave C++ libraries use different and incompatible exceptions support schemes. As such, you must specify `-fno-exceptions` when using `-stdlib=legacy_cpplib`.

Related references

1.36 -stdlib on page 1-51.

Related information

Standard C++ library implementation definition.

1.12 -fshort-enums, -fno-short-enums

Allows the compiler to set the size of an enumeration type to the smallest data type that can hold all enumerator values.

The `-fshort-enums` option can improve memory usage, but might reduce performance because narrow memory accesses can be less efficient than full register-width accesses.

Note

All linked objects, including libraries, must make the same choice. It is not possible to link an object file compiled with `-fshort-enums`, with another object file that is compiled without `-fshort-enums`.

Note

The `-fshort-enums` option is not supported for AArch64. The *Procedure Call Standard for the ARM® 64-bit Architecture* mandates that the size of enumeration types must be at least 32 bits. If the `-fshort-enums` option is specified for an AArch64 target, it is ignored.

Default

The default is `-fno-short-enums`. That is, the size of an enumeration type is at least 32 bits regardless of the size of the enumerator values.

Example

This example shows the size of four different enumeration types: 8-bit, 16-bit, 32-bit, and 64-bit integers.

```
#include <stdio.h>

// Largest value is 8-bit integer
enum int8Enum  {int8Val1 =0x01, int8Val2 =0x02, int8Val3 =0xF1 };

// Largest value is 16-bit integer
enum int16Enum {int16Val1=0x01, int16Val2=0x02, int16Val3=0xFFF1 };

// Largest value is 32-bit integer
enum int32Enum {int32Val1=0x01, int32Val2=0x02, int32Val3=0xFFFFFFF1 };

// Largest value is 64-bit integer
enum int64Enum {int64Val1=0x01, int64Val2=0x02, int64Val3=0xFFFFFFFFFFFFFFF1 };

int main(void)
{
  printf("size of int8Enum  is %zd\n", sizeof (enum int8Enum));
  printf("size of int16Enum is %zd\n", sizeof (enum int16Enum));
  printf("size of int32Enum is %zd\n", sizeof (enum int32Enum));
  printf("size of int64Enum is %zd\n", sizeof (enum int64Enum));
}
```

When compiled without the `-fshort-enums` option, all enumeration types are 32 bits (4 bytes) except for `int64Enum` which requires 64 bits (8 bytes):

```
armclang --target=arm-arm-eabi-none -march=armv8-a enum_test.cpp

size of int8Enum  is 4
size of int16Enum is 4
size of int32Enum is 4
size of int64Enum is 8
```

When compiled with the `-fshort-enums` option, each enumeration type has the smallest size possible to hold the largest enumerator value:

```
armclang -fshort-enums --target=arm-arm-eabi-none -march=armv8-a enum_test.cpp

size of int8Enum  is 1
size of int16Enum is 2
size of int32Enum is 4
size of int64Enum is 8
```

Note

ISO C restricts enumerator values to the range of `int`. By default `armclang` does not issue warnings about enumerator values that are too large, but with `-Wpedantic` a warning is displayed.

Related information

Procedure Call Standard for the ARM 64-bit Architecture (AArch64).

1.13 -fshort-wchar, -fno-short-wchar

Sets the size of `wchar_t` to 2 bytes.

The `-fshort-wchar` option can improve memory usage, but might reduce performance because narrow memory accesses can be less efficient than full register-width accesses.

Note

All linked objects must use the same `wchar_t` size, including libraries. It is not possible to link an object file compiled with `-fshort-wchar`, with another object file that is compiled without `-fshort-wchar`.

Default

The default is `-fno-short-wchar`. That is. the default size of `wchar_t` is 4 bytes.

Example

This example shows the size of the `wchar_t` type:

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

int main(void)
{
  printf("size of wchar_t is %zd\n", sizeof (wchar_t));
  return 0;
}
```

When compiled without the `-fshort-wchar` option, the size of `wchar_t` is 4 bytes:

```
armclang --target=aarch64-arm-none-eabi wchar_test.c

size of wchar_t is 4
```

When compiled with the `-fshort-wchar` option, the size of `wchar_t` is 2 bytes:

```
armclang -fshort-wchar --target=aarch64-arm-none-eabi wchar_test.c

size of wchar_t is 2
```

1.14 -fvectorize, -fno-vectorize

Enables and disables the generation of Advanced SIMD vector instructions directly from C or C++ code at optimization levels `-O1` and higher.

Note

The `-fvectorize` option is not supported for AArch64 state. The compiler never performs automatic vectorization for AArch64 state targets.

Default

The default depends on the optimization level in use.

At optimization level `-O0` (the default optimization level), `armclang` never performs automatic vectorization. The `-fvectorize` and `-fno-vectorize` options are ignored.

At optimization level `-O1`, the default is `-fno-vectorize`. Use `-fvectorize` to enable automatic vectorization.

At optimization level `-O2` and above, the default is `-fvectorize`. Use `-fno-vectorize` to disable automatic vectorization.

Example

This example enables automatic vectorization with optimization level `-O1`:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -fvectorize -O1 -c file.c
```

Related references

1.1 -c on page 1-14.
1.32 -O on page 1-47.

1.15 -g, -gdwarf-2, -gdwarf-3, -gdwarf-4,

Adds debug tables for source-level debugging.

Syntax

`-g`

`-gdwarf-`*`version`*

Where:

`version`
is the DWARF format to produce. Valid values are 2, 3, and 4.

The `-g` option is a synonym for `-gdwarf-4`.

Usage

The compiler produces debug information that is compatible with the specified DWARF standard.

Use a compatible debugger to load, run, and debug images. For example, ARM DS-5 Debugger is compatible with DWARF 4. Compile with the `-g` or `-gdwarf-4` options to debug with ARM DS-5 Debugger.

Legacy and third-party tools might not support DWARF 4 debug information. In this case you can specify the level of DWARF conformance required using the `-gdwarf-2` or `-gdwarf-3` options.

Because the DWARF 4 specification supports language features that are not available in earlier versions of DWARF, the `-gdwarf-2` and `-gdwarf-3` options should only be used for backwards compatibility.

Default

By default, `armclang` does not produce debug information.

Examples

If you specify multiple options, the last option specified takes precedence. For example:

- `-gdwarf-3 -gdwarf-2` produces DWARF 2 debug, because `-gdwarf-2` overrides `-gdwarf-3`.
- `-g -gdwarf-2` produces DWARF 2 debug, because `-gdwarf-2` overrides `-g` (a synonym for `-gdwarf-4`).
- `-gdwarf-2 -g` produces DWARF 4 debug, because `-g` (a synonym for `-gdwarf-4`) overrides `-gdwarf-2`.

1.16 -I

Adds the specified directory to the list of places that are searched to find included files.

If you specify more than one directory, the directories are searched in the same order as the `-I` options specifying them.

Syntax

`-I`*`dir`*

Where:

`dir`
is a directory to search for included files.

Use multiple `-I` options to specify multiple search directories.

1.17 -L

Specifies a list of paths that the linker searches for user libraries.

Syntax

`-L dir[,dir,...]`

Where:

dir`[,dir,...]`
is a comma-separated list of directories to be searched for user libraries.

At least one directory must be specified.

When specifying multiple directories, do not include spaces between commas and directory names in the list.

`armclang` translates this option to `--userlibpath` and passes it to `armlink`.

See the *ARM® Compiler toolchain Linker Reference* for information about the `--userlibpath` linker option.

Note

The `-L` option has no effect when used with the `-c` option, that is when not linking.

Related information

ARM Compiler toolchain Linker Reference.

1.18 -l

Add the specified library to the list of searched libraries.

Syntax

`-l` *name*

Where *name* is the name of the library.

`armclang` translates this option to `--library` and passes it to `armlink`.

See the *ARM® Compiler toolchain Linker Reference* for information about the `--library` linker option.

Note

The `-l` option has no effect when used with the `-c` option, that is when not linking.

Related information

ARM Compiler toolchain Linker Reference.

1.19 -M

Produces a list of makefile dependency rules suitable for use by a make utility.

The compiler executes only the preprocessor step of the compilation. By default, output is on the standard output stream.

If you specify multiple source files, a single dependency file is created.

——— **Note** ———

The `-MT` option lets you override the target name in the dependency rules.

———————————

——— **Note** ———

The `-MD` option lets you compile the source files as well as produce makefile dependency rules.

———————————

Example

You can redirect output to a file using standard UNIX and MS-DOS notation, the `-o` option, or the `-MF` option. For example:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -M source.c > deps.mk
armclang --target=arm-arm-none-eabi -march=armv8-a -M source.c -o deps.mk
armclang --target=arm-arm-none-eabi -march=armv8-a -M source.c -MF deps.mk
```

Related references

1.31 -o on page 1-46.
1.20 -MD on page 1-34.
1.21 -MF on page 1-35.
1.22 -MT on page 1-36.

1.20 -MD

Compiles source files and produces a list of makefile dependency rules suitable for use by a make utility.

The compiler creates a makefile dependency file for each source file, using a `.d` suffix.

Example

The following example creates makefile dependency lists `test1.d` and `test2.d` and compiles the source files to an image with the default name, `a.out`:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -MD test1.c test2.c
```

Related references

1.19 -M on page 1-33.
1.21 -MF on page 1-35.
1.22 -MT on page 1-36.

1.21 -MF

Specifies a filename for the makefile dependency rules produced by the `-M` and `-MD` options.

Syntax

`-MF` *file*

Where:

file
Specifies the filename for the makefile dependency rules.

Note

The `-MF` option only has an effect when used in conjunction with either the `-M` or `-MD` options.

`armclang -M` sends output to the standard output stream by default. The `-MF` option sends output to the specified filename instead.

If you specify multiple source files with `-M`, makefile dependency rules for all source files are concatenated. If you also specify `-MF`, the concatenated dependency rules are saved to the specified file.

`armclang -MD` sends output to a file with the same name as the source file by default, but with a `.d` suffix. The `-MF` option sends output to the specified filename instead. Only use a single source file with `armclang -MD -MF`.

Examples

This example sends makefile dependency rules to standard output, without compiling the source:

```
armclang --target=aarch64-arm-none-eabi -M source.c
```

This example saves makefile dependency rules to `deps.mk`, without compiling the source:

```
armclang --target=aarch64-arm-none-eabi -M source.c -MF deps.mk
```

This example sends concatenated makefile dependency rules for multiple source files to standard output, without compiling the source:

```
armclang --target=aarch64-arm-none-eabi -M source1.c source2.c source3.c
```

This example saves concatenated makefile dependency rules for multiple source files to `deps.mk`, without compiling the source:

```
armclang --target=aarch64-arm-none-eabi -M source1.c source2.c source3.c -MF deps.mk
```

This example compiles the source and saves makefile dependency rules to `source.d` (using the default file naming rules):

```
armclang --target=aarch64-arm-none-eabi -MD source.c
```

This example compiles the source and saves makefile dependency rules to `deps.mk`:

```
armclang --target=aarch64-arm-none-eabi -MD source.c -MF deps.mk
```

Related references

1.19 -M on page 1-33.
1.20 -MD on page 1-34.
1.22 -MT on page 1-36.

1.22 -MT

Changes the target of the makefile dependency rule produced by `-M`.

Note

The `-MT` option only has an effect when used in conjunction with either the `-M` or `-MD` options.

By default, `armclang -M` creates makefile dependencies rules based on the source filename:

```
armclang --target=aarch64-arm-none-eabi -M test.c
test.o: test.c header.h
```

The `-MT` option renames the target of the makefile dependency rule:

```
armclang --target=aarch64-arm-none-eabi -M test.c -MT foo
foo: test.c header.h
```

The compiler executes only the preprocessor step of the compilation. By default, output is on the standard output stream.

If you specify multiple source files, the `-MT` option renames the target of all dependency rules:

```
armclang --target=aarch64-arm-none-eabi -M test1.c test2.c -MT foo
foo: test1.c header.h
foo: test2.c header.h
```

Specifying multiple `-MT` options creates multiple targets for each rule:

```
armclang --target=aarch64-arm-none-eabi -M test1.c test2.c -MT foo -MT bar
foo bar: test1.c header.h
foo bar: test2.c header.h
```

Related references

1.19 -M on page 1-33.
1.20 -MD on page 1-34.
1.21 -MF on page 1-35.

1.23 -march

Targets an architecture profile, generating generic code that runs on any processor of that architecture.

Syntax

`-march=`*name*

Where:

name

Specifies the architecture.

The following are valid `-march` values:

`armv8-a`

ARMv8-A architecture profile. Valid with both `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`.

`armv7-a`

ARMv7-A architecture profile. Only valid with `--target=arm-arm-none-eabi`.

Default

For AArch64 targets (`--target=aarch64-arm-none-eabi`), unless you target a particular processor using `-mcpu`. the compiler defaults to `-march=armv8-a`, generating generic code for ARMv8-A in AArch64 state.

For AArch32 targets (`--target=arm-arm-none-eabi`) there is no default. You must specify either `-march` (to target an architecture) or `-mcpu` (to target a processor).

1.24 -marm

Requests that the compiler targets the A32 or ARM instruction sets.

Different architectures support different instruction sets:

- ARMv8-A processors in AArch64 state execute A64 instructions.
- ARMv8-A processors in AArch32 state can execute A32 or T32 instructions.
- ARMv7-A processors can execute ARM or Thumb instructions.

The `-marm` option targets the A32 (ARMv8-A AArch32 state) or ARM (ARMv7-A) instruction set. This is the default for the `arm-arm-none-eabi` target.

Note

The `-marm` option is not valid with AArch64 targets, for example `--target=aarch64-arm-none-eabi`. The compiler ignores the `-marm` option and generates a warning with AArch64 targets.

Default

The default for ARMv8-A AArch32 and ARMv7-A targets is `-marm`.

Related references

1.30 -mthumb on page 1-45.

1.37 --target on page 1-52.

1.26 -mcpu on page 1-40.

Related information

Specifying a target architecture, processor, and instruction set.

1.25 -mbig-endian

Generates code suitable for an ARM processor using byte-invariant big-endian (BE-8) data.

Default

The default is `-mlittle-endian`.

Related references

1.29 -mlittle-endian on page 1-44.

1.26 -mcpu

Enables code generation for a specific ARM processor.

Syntax

`-mcpu=`*`name`*

`-mcpu=`*`name`*`[+[no]`*`feature`*`]*` (AArch64 targets only)

Where:

`name`

Specifies the processor.

The following are valid `-mcpu` values with `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`:

- `cortex-a53`
- `cortex-a57`
- `cortex-a72`

The following are valid `-mcpu` values with `--target=arm-arm-none-eabi` only:

- `cortex-a5`
- `cortex-a7`
- `cortex-a8`
- `cortex-a9`
- `cortex-a12`
- `cortex-a15`
- `cortex-a17`

`feature`

Enables or disables an optional architectural feature (AArch64 targets only), any of the following:

- `crc` - enable CRC instructions.
- `crypto` - enable the cryptographic extension.
- `fp` - enable the floating-point extension.
- `simd` - enable the NEON advanced SIMD extension.

Note

`name` and *`feature`* are case-sensitive.

Usage

For AArch64 targets only, you can use `-mcpu` option to enable and disable specific architectural features.

To disable a feature, prefix with `no`, for example `cortex-a57+nocrypto`.

To enable or disable multiple features, chain multiple feature modifiers. For example, to enable CRC instructions and disable all other extensions:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57+nocrypto+nofp+nosimd+crc
```

If you specify conflicting feature modifiers with `-mcpu`, the rightmost feature is used. For example, the following command enables the floating-point extension:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57+nofp+fp
```

You can prevent the use of floating-point instructions or floating-point registers for AArch64 targets with the `-mcpu=`*`name`*`+nofp+nosimd` option. Subsequent use of floating-point data types in this mode is unsupported.

Default

For AArch64 targets (`--target=aarch64-arm-none-eabi`), the compiler generates generic code for the ARMv8-A architecture in AArch64 state by default.

For AArch32 targets (`--target=arm-arm-none-eabi`) there is no default. You must specify either `-march` (to target an architecture) or `-mcpu` (to target a processor).

Examples

To target the AArch64 state of a Cortex®-A57 processor:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 test.c
```

To target the AArch32 state of a Cortex-A53 processor, generating A32 instructions:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a53 -marm test.c
```

Related references

1.28 -mfpu on page 1-43.
1.24 -marm on page 1-38.
1.30 -mthumb on page 1-45.
1.37 --target on page 1-52.
1.28 -mfpu on page 1-43.
1.37 --target on page 1-52.

Related information

Specifying a target architecture, processor, and instruction set.
Preventing the use of floating-point instructions and registers.

1.27 -mfloat-abi

Specifies whether to use hardware instructions or software library functions for floating-point operations, and which registers are used to pass floating-point parameters and return values.

Syntax

`-mfloat-abi=`*`value`*

Where *`value`* is one of:

`soft`
: Software library functions for floating-point operations and software floating-point linkage.

`softfp`
: Hardware floating-point instructions and software floating-point linkage.

`hard`
: Hardware floating-point instructions and hardware floating-point linkage.

Note

The `-mfloat-abi` option is not valid with ARMv8 AArch64 targets. AArch64 targets use hardware floating-point instructions and hardware floating-point linkage. However, you can prevent the use of floating-point instructions or floating-point registers for AArch64 targets with the `-mcpu=`*`name`*`+nofp +nosimd` option. Subsequent use of floating-point data types in this mode is unsupported.

Default

The default for `--target=arm-arm-none-eabi` is `softfp`.

Related references

1.28 -mfpu on page 1-43.

1.28 -mfpu

Specifies the target FPU architecture, that is the floating-point hardware available on the target.

Syntax

`-mfpu=`*name*

Where *name* is one of the following:

`vfpv3`
: Enable the ARMv7 VFPv3 floating-point extension. Disable the Advanced SIMD extension.

`neon-vfpv3`
: Enable the ARMv7 VFPv3 floating-point extension and the Advanced SIMD extension.

`vfpv4`
: Enable the ARMv7 VFPv4 floating-point extension. Disable the Advanced SIMD extension.

`neon-vfpv4`
: Enable the ARMv7 VFPv4 floating-point extension and the Advanced SIMD extension.

`fp-armv8`
: Enable the ARMv8 floating-point extension. Disable the cryptographic extension and the Advanced SIMD extension.

`neon-fp-armv8`
: Enable the ARMv8 floating-point extension and the Advanced SIMD extensions. Disable the cryptographic extension.

`crypto-neon-fp-armv8`
: Enable the ARMv8 floating-point extension, the cryptographic extension. and the Advanced SIMD extension.

The `-mfpu` option overrides the default FPU option implied by the target architecture.

Note

- The `-mfpu` option is ignored with AArch64 targets, for example `aarch64-arm-none-eabi`. Use the `-mcpu` option to override the default FPU for `aarch64-arm-none-eabi` targets. For example, to prevent the use of floating-point instructions or floating-point registers for the `aarch64-arm-none-eabi` target use the `-mcpu=`*name*`+nofp+nosimd` option. Subsequent use of floating-point data types in this mode is unsupported.
- The `-mfpu` option is also ignored if `-mfloat-abi=soft`, which is the default for `arm-arm-none-eabi`.
- In ARMv7, the Advanced SIMD extension was called the NEON Advanced SIMD extension.

Default

The default FPU option depends on the target architecture.

Related references

1.26 -mcpu on page 1-40.
1.27 -mfloat-abi on page 1-42.
1.26 -mcpu on page 1-40.
1.37 --target on page 1-52.

Related information

Specifying a target architecture, processor, and instruction set.
Preventing the use of floating-point instructions and registers.

1.29 -mlittle-endian

Generates code suitable for an ARM processor using little-endian data.

Default

The default is `-mlittle-endian`.

Related references

1.25 -mbig-endian on page 1-39.

1.30 -mthumb

Requests that the compiler targets the T32 or Thumb® instruction sets.

Different architectures support different instruction sets:

- ARMv8-A processors in AArch64 state execute A64 instructions.
- ARMv8-A processors in AArch32 state can execute A32 or T32 instructions.
- ARMv7-A processors can execute ARM or Thumb instructions.

The `-mthumb` option targets the T32 (ARMv8-A AArch32 state) or Thumb (ARMv7-A) instruction set.

Note

The `-mthumb` option is not valid with AArch64 targets, for example `--target=aarch64-arm-none-eabi`. The compiler ignores the `-mthumb` option and generates a warning with AArch64 targets.

Default

The default for ARMv8-A AArch32 and ARMv7-A targets is `-marm`.

Example

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -mthumb test.c
```

Related references

1.24 -marm on page 1-38.
1.37 --target on page 1-52.
1.26 -mcpu on page 1-40.

Related information

Specifying a target architecture, processor, and instruction set.

1.31 -o

Specifies the name of the output file.

The option `-o filename` specifies the name of the output file produced by the compiler.

The option `-o-` redirects output to the standard output stream when used with the `-c` or `-S` options.

Default

If you do not specify a `-o` option, the compiler names the output file according to the conventions described by the following table.

Table 1-1 Compiling without the -o option

Compiler option	Action	Usage notes
`-c`	Produces an object file whose name defaults to the name of the input file with the filename extension `.o`	
`-S`	Produces an output file whose name defaults to the name of the input file with the filename extension `.s`	
`-E`	Writes output from the preprocessor to the standard output stream	
(No option)	Produces temporary object files, then automatically calls the linker to produce an executable image with the default name of `a.out`	None of `-o`, `-c`, `-E` or `-S` is specified on the command line

1.32 -O

Specifies the level of optimization to use when compiling source files.

Syntax

`-O`*level*

Where *level* is one of the following:

0

Minimum optimization. Turns off most optimizations. When debugging is enabled, this option gives the best possible debug view because the structure of the generated code directly corresponds to the source code.

This is the default optimization level.

1

Restricted optimization. When debugging is enabled, this option gives a generally satisfactory debug view with good code density.

2

High optimization. When debugging is enabled, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The compiler might perform optimizations that cannot be described by debug information.

3

Maximum optimization. When debugging is enabled, this option typically gives a poor debug view. ARM recommends debugging at lower optimization levels.

fast

Enables all the optimizations from `-O3` along with other aggressive optimizations that might violate strict compliance with language standards.

s

Performs optimizations to reduce code size, balancing code size against code speed.

z

Performs optimizations to minimize image size.

Default

If you do not specify `-O`*level*, the compiler assumes `-O0`.

1.33 -rdynamic

If an executable has dynamic symbols, export all externally visible symbols rather than only referenced symbols.

`armclang` translates this option to `--export-dynamic` and passes it to `armlink`.

See the *ARM® Compiler toolchain Linker Reference* for information about the `--export-dynamic` linker option.

Related information

ARM Compiler toolchain Linker Reference.

1.34 -S

Outputs the disassembly of the machine code generated by the compiler.

Object modules are not generated. The name of the assembly output file defaults to *filename*`.s` in the current directory, where *filename* is the name of the source file stripped of any leading directory names. The default filename can be overridden with the `-o` option.

Related references

1.31 -o on page 1-46.

1.35 -std

Specifies the language standard to compile for.

Syntax

`-std=`*name*

Where:

name

Specifies the language mode. Valid values include:

c90
C as defined by the 1990 C standard.

gnu90
C as defined by the 1990 C standard, with additional GNU extensions.

c99
C as defined by the 1999 C standard.

gnu99
C as defined by the 1999 C standard, with additional GNU extensions.

c11
C as defined by the 2011 C standard.

gnu11
C as defined by the 2011 C standard, with additional GNU extensions.

c++98
C++ as defined by the 1998 standard.

gnu++98
C++ as defined by the 1998 standard, with additional GNU extensions.

c++11
C++ as defined by the 2011 standard.

gnu++11
C++ as defined by the 2011 standard, with additional GNU extensions.

For C++ code, the default is `gnu++98`. For more information about C++ support, see *C++ Status* on the Clang web site.

For C code, the default is `gnu11`. For more information about C support, see *Language Compatibility* on the Clang web site.

Related references

1.45 -x on page 1-60.

Related information

Language Compatibility.
C++ Status.

1.36 -stdlib

Specifies the C++ Library to add to the system include path.

Syntax

`-stdlib=`*`library_option`*

Where *`library_option`* is one of the following:

`libc++`
The default library. Adds `include/libcxx` to the C++ system include path.

`legacy_cpplib`
Adds `include/cpplib` to the C++ system include path. This option is for the C++ Rogue Wave libraries.

Note

- When performing a compile and link as a single operation, `armclang` automatically passes `--stdlib=`*`library_option`* to `armlink`.
- The clang option `-stdlib=libstdc++` is unsupported.

Related information

--stdlib.

1.37 --target

Generate code for the specified target triple.

Syntax

`--target=` *`triple`*

Where:

`triple`
has the form *`architecture-vendor-OS-abi`*.

Supported targets are as follows:

`aarch64-arm-none-eabi`
Generates A64 instructions for AArch64 state. Implies `-march=armv8-a` unless `-mcpu` is specified.

`arm-arm-none-eabi`
Generates A32/T32 instructions for AArch32 state. Must be used in conjunction with `-march` (to target an architecture) or `-mcpu` (to target a processor).

Note

- The targets are case-sensitive.
- The `--target` option is an `armclang` option. For all of the other tools, such as `armasm` and `armlink`, use the `--cpu` and `--fpu` options to specify target processors and architectures.

Default

The `--target` option is mandatory and has no default. You must always specify a target triple.

Related references

1.24 -marm on page 1-38.
1.30 -mthumb on page 1-45.
1.26 -mcpu on page 1-40.
1.26 -mcpu on page 1-40.
1.28 -mfpu on page 1-43.

Related information

Specifying a target architecture, processor, and instruction set.
armasm User Guide.
armlink User Guide.

1.38 -u

Prevents the removal of a specified symbol if it is undefined.

Syntax

`-u` *`symbol`*

Where *`symbol`* is the symbol to keep.

`armclang` translates this option to `--undefined` and passes it to `armlink`.

See the *ARM® Compiler toolchain Linker Reference* for information about the `--undefined` linker option.

Related information

ARM Compiler toolchain Linker Reference.

1.39 -v

Displays the commands that invoke the compiler and linker, and executes those commands.

Usage

The `-v` compiler option produces diagnostic output showing exactly how the compiler and linker are invoked, displaying the options for each tool. The `-v` compiler option also displays version information.

With the `-v` option, `armclang` displays this diagnostic output and executes the commands.

Note

To display the diagnostic output without executing the commands, use the `-###` option.

Related references

1.46 -### on page 1-61.

1.40 --version

Displays version information.

1.41 --version_number

Displays the version of `armclang` you are using.

Usage

The compiler displays the version number in the format `nnnbbbb`, where:

- `nnn` is the version number.
- `bbbb` is the build number.

Related references

5.1 Predefined macros on page 5-114.

1.42 -W

Controls diagnostics.

Syntax

`-W`*`name`*

Where common values for *`name`* include:

`-Werror`
: Turn warnings into errors.

`-Werror=foo`
: Turn warning `foo` into an error.

`-Wno-error=foo`
: Leave warning `foo` as a warning even if `-Werror` is specified.

`-Wfoo`
: Enable warning `foo`.

`-Wno-foo`
: Suppress warning `foo`.

`-Weverything`
: Enable all warnings.

See *Controlling Errors and Warnings* in the *Clang Compiler User's Manual* for full details about controlling diagnostics with `armclang`.

Related information

Options for controlling diagnostics with armclang.

1.43 -Wl

Specifies command-line options to pass to the linker when a link step is being performed after compilation.

See the *ARM® Compiler toolchain Linker Reference* for information about available linker options.

Syntax

`-Wl,opt,[opt[,...]]`

Where:

opt
is a command-line option to pass to the linker.

You can specify a comma-separated list of options or `option=argument` pairs.

Restrictions

The linker generates an error if `-Wl` passes unsupported options.

Examples

The following examples show the different syntax usages. They are equivalent because `armlink` treats the single option `--list=diag.txt` and the two options `--list diag.txt` equivalently:

```
armclang --target=aarch64-arm-none-eabi hello.c -Wl,--split,--list,diag.txt
armclang --target=aarch64-arm-none-eabi hello.c -Wl,--split,--list=diag.txt
```

Related references

1.44 -Xlinker on page 1-59.

1.44 -Xlinker

Specifies command-line options to pass to the linker when a link step is being performed after compilation.

See the *ARM® Compiler toolchain Linker Reference* for information about available linker options.

Syntax

`-Xlinker` *opt*

Where:

opt
is a command-line option to pass to the linker.

If you want to pass multiple options, use multiple `-Xlinker` options.

Restrictions

The linker generates an error if `-Xlinker` passes unsupported options.

Examples

This example passes the option `--split` to the linker:

```
armclang --target=aarch64-arm-none-eabi hello.c -Xlinker --split
```

This example passes the options `--list diag.txt` to the linker:

```
armclang --target=aarch64-arm-none-eabi hello.c -Xlinker --list -Xlinker diag.txt
```

Related references

1.43 -Wl on page 1-58.

1.45 -x

Specifies the language of source files.

Syntax

`-x` *Language*

Where:

Language
Specifies the language of subsequent source files, one of the following:

`c`
C code.

`c++`
C++ code.

`assembler-with-cpp`
Assembly code containing C directives that require the C preprocessor.

`assembler`
Assembly code that does not require the C preprocessor.

Usage

`-x` overrides the default language standard for the subsequent input files that follow it on the command-line. For example:

```
armclang inputfile1.s -xc inputfile2.s inputfile3.s
```

In this example, `armclang` treats the input files as follows:

- `inputfile1.s` appears before the `-xc` option, so `armclang` treats it as assembly code because of the `.s` suffix.
- `inputfile2.s` and `inputfile3.s` appear after the `-xc` option, so `armclang` treats them as C code.

Note

Use `-std` to set the default language standard.

Default

By default the compiler determines the source file language from the filename suffix, as follows:

- `.cpp`, `.cxx`, `.c++`, `.cc`, and `.CC` indicate C++, equivalent to `-x c++`.
- `.c` indicates C, equivalent to `-x c`.
- `.s` (lower-case) indicates assembly code that does not require preprocessing, equivalent to `-x assembler`.
- `.S` (upper-case) indicates assembly code that requires preprocessing, equivalent to `-x assembler-with-cpp`.

Related references

1.35 -std on page 1-50.

1.46 -###

Displays the commands that invoke the compiler and linker, without executing those commands.

Usage

The `-###` compiler option produces diagnostic output showing exactly how the compiler and linker are invoked, displaying the options for each tool. The `-###` compiler option also displays version information.

With the `-###` option, `armclang` only displays this diagnostic output. `armclang` does not compile source files or invoke `armlink`.

Note

To display the diagnostic output and execute the commands, use the `-v` option.

Related references

1.39 -v on page 1-54.

Chapter 2
Compiler-specific Keywords and Operators

Summarizes the compiler-specific keywords and operators that are extensions to the C and C++ Standards.

It contains the following sections:

2.1 Compiler-specific keywords and operators

The ARM compiler `armclang` provides keywords that are extensions to the C and C++ Standards.

Standard C and Standard C++ keywords that do not have behavior or restrictions specific to the ARM compiler are not documented.

Keyword extensions that the ARM compiler supports:

- `__alignof__`
- `__asm`
- `__declspec`
- `__inline`

Related references

2.2 __alignof__ on page 2-64.
2.3 __asm on page 2-66.
2.4 __declspec attributes on page 2-67.
2.8 __inline on page 2-71.

2.2 __alignof__

The `__alignof__` keyword enables you to enquire about the alignment of a type or variable.

——— **Note** ———

This keyword is a GNU compiler extension that the ARM compiler supports.

———————————

Syntax

`__alignof__(`*type*`)`

`__alignof__(`*expr*`)`

Where:

type
is a type

expr
is an lvalue.

Return value

`__alignof__(`*type*`)` returns the alignment requirement for the type, or 1 if there is no alignment requirement.

`__alignof__(`*expr*`)` returns the alignment requirement for the type of the lvalue *expr*, or 1 if there is no alignment requirement.

Example

The following example displays the alignment requirements for a variety of data types, first directly from the data type, then from an lvalue of the corresponding data type:

```
#include <stdio.h>

int main(void)
{
  int       var_i;
  char      var_c;
  double    var_d;
  float     var_f;
  long      var_l;
  long long var_ll;

  printf("Alignment requirement from data type:\n");
  printf("  int       : %d\n", __alignof__(int));
  printf("  char      : %d\n", __alignof__(char));
  printf("  double    : %d\n", __alignof__(double));
  printf("  float     : %d\n", __alignof__(float));
  printf("  long      : %d\n", __alignof__(long));
  printf("  long long : %d\n", __alignof__(long long));
  printf("\n");
  printf("Alignment requirement from data type of lvalue:\n");
  printf("  int       : %d\n", __alignof__(var_i));
  printf("  char      : %d\n", __alignof__(var_c));
  printf("  double    : %d\n", __alignof__(var_d));
  printf("  float     : %d\n", __alignof__(var_f));
  printf("  long      : %d\n", __alignof__(var_l));
  printf("  long long : %d\n", __alignof__(var_ll));
}
```

Compiling with the following command produces the following output:

```
armclang --target=arm-arm-none-eabi -march=armv8-a alignof_test.c -o alignof.axf
```

```
Alignment requirement from data type:
  int       : 4
  char      : 1
  double    : 8
  float     : 4
```

```
  long      : 4
  long long : 8

Alignment requirement from data type of lvalue:
  int       : 4
  char      : 1
  double    : 8
  float     : 4
  long      : 4
  long long : 8
```

2.3 __asm

This keyword passes information to the `armclang` assembler.

The precise action of this keyword depends on its usage.

Usage

Inline assembly

The `__asm` keyword can incorporate inline GCC syntax assembly code into a function. For example:

```
#include <stdio.h>

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

int main(void)
{
  int a = 1;
  int b = 2;
  int c = 0;

  c = add(a,b);

  printf("Result of %d + %d = %d\n", a, b, c);
}
```

The general form of an `__asm` inline assembly statement is:

`__asm(`*`code`* `[: `*`output_operand_list`* `[: `*`input_operand_list`* `[: `*`clobbered_register_list`*`]]]);`

`code` is the assembly code. In our example, this is `"ADD %[result], %[input_i], %[input_j]"`.

`output_operand_list` is an optional list of output operands, separated by commas. Each operand consists of a symbolic name in square brackets, a constraint string, and a C expression in parentheses. In our example, there is a single output operand: `[result] "=r" (res)`.

`input_operand_list` is an optional list of input operands, separated by commas. Input operands use the same syntax as output operands. In our example there are two input operands: `[input_i] "r" (i), [input_j] "r" (j)`.

`clobbered_register_list` is an optional list of clobbered registers. In our example, this is omitted.

Assembly labels

The `__asm` keyword can specify an assembly label for a C symbol. For example:

```
int count __asm__("count_v1"); // export count_v1, not count
```

2.4 __declspec attributes

The `__declspec` keyword enables you to specify special attributes of objects and functions.

The `__declspec` keyword must prefix the declaration specification. For example:

```
__declspec(noreturn) void overflow(void);
```

The available `__declspec` attributes are as follows:

- `__declspec(noinline)`
- `__declspec(noreturn)`
- `__declspec(nothrow)`

`__declspec` attributes are storage class modifiers. They do not affect the type of a function or variable.

Related references

2.5 __declspec(noinline) on page 2-68.
2.6 __declspec(noreturn) on page 2-69.
2.7 __declspec(nothrow) on page 2-70.

2.5 __declspec(noinline)

The `__declspec(noinline)` attribute suppresses the inlining of a function at the call points of the function.

`__declspec(noinline)` can also be applied to constant data, to prevent the compiler from using the value for optimization purposes, without affecting its placement in the object. This is a feature that can be used for patchable constants, that is, data that is later patched to a different value. It is an error to try to use such constants in a context where a constant value is required. For example, an array dimension.

Note

This `__declspec` attribute has the function attribute equivalent `__attribute__((noinline))`.

Example

```
/* Prevent y being used for optimization */
__declspec(noinline) const int y = 5;
/* Suppress inlining of foo() wherever foo() is called */
__declspec(noinline) int foo(void);
```

2.6 __declspec(noreturn)

The `__declspec(noreturn)` attribute asserts that a function never returns.

Note

This `__declspec` attribute has the function attribute equivalent `__attribute__((noreturn))`.

Usage

Use this attribute to reduce the cost of calling a function that never returns, such as `exit()`. If a `noreturn` function returns to its caller, the behavior is undefined.

Restrictions

The return address is not preserved when calling the `noreturn` function. This limits the ability of a debugger to display the call stack.

Example

```
__declspec(noreturn) void overflow(void); // never return on overflow
int negate(int x)
{
    if (x == 0x80000000) overflow();
    return -x;
}
```

2.7 __declspec(nothrow)

The `__declspec(nothrow)` attribute asserts that a call to a function never results in a C++ exception being propagated from the callee into the caller.

The ARM library headers automatically add this qualifier to declarations of C functions that, according to the ISO C Standard, can never throw an exception. However, there are some restrictions on the unwinding tables produced for the C library functions that might throw an exception in a C++ context, for example, `bsearch` and `qsort`.

Note

This `__declspec` attribute has the function attribute equivalent `__attribute__((nothrow))`.

Usage

If the compiler knows that a function can never throw an exception, it might be able to generate smaller exception-handling tables for callers of that function.

Restrictions

If a call to a function results in a C++ exception being propagated from the callee into the caller, the behavior is undefined.

This modifier is ignored when not compiling with exceptions enabled.

Example

```
struct S
{
    ~S();
};
__declspec(nothrow) extern void f(void);
void g(void)
{
    S s;
    f();
}
```

Related information

Standard C++ library implementation definition.

2.8 __inline

The `__inline` keyword suggests to the compiler that it compiles a C or C++ function inline, if it is sensible to do so.

`__inline` can be used in C90 code, and serves as an alternative to the C99 `inline` keyword.

Both `__inline` and `__inline__` are supported in `armclang`.

Example

```
static __inline int f(int x){
    return x*5+1;
}

int g(int x, int y){
    return f(x) + f(y);
}
```

Related concepts

5.2 Inline functions on page 5-117.

Chapter 3
Compiler-specific Function, Variable, and Type Attributes

Summarizes the compiler-specific function, variable, and type attributes that are extensions to the C and C++ Standards.

It contains the following sections:

3.1 Function attributes

The `__attribute__` keyword enables you to specify special attributes of variables, structure fields, functions, and types.

The keyword format is either of the following:

```
__attribute__((attribute1, attribute2, ...))
__attribute__((__attribute1__, __attribute2__, ...))
```

For example:

```
int my_function(int b) __attribute__((const));
static int my_variable __attribute__((__unused__));
```

The following table summarizes the available function attributes.

Table 3-1 Function attributes that the compiler supports, and their equivalents

Function attribute	Non-attribute equivalent
`__attribute__((alias))`	-
`__attribute__((always_inline))`	-
`__attribute__((const))`	-
`__attribute__((constructor[(priority)]))`	-
`__attribute__((deprecated))`	-
`__attribute__((destructor[(priority)]))`	-
`__attribute__((format_arg(string-index)))`	-
`__attribute__((malloc))`	-
`__attribute__((noinline))`	`__declspec(noinline)`
`__attribute__((nomerge))`	-
`__attribute__((nonnull))`	-
`__attribute__((noreturn))`	`__declspec(noreturn))`
`__attribute__((nothrow))`	`__delspec(nothrow)`
`__attribute__((notailcall))`	-
`__attribute__((pcs("calling_convention")))`	-
`__attribute__((pure))`	-
`__attribute__((section("name")))`	-
`__attribute__((unused))`	-
`__attribute__((used))`	-
`__attribute__((visibility("visibility_type")))`	-
`__attribute__((weak))`	-
`__attribute__((weakref("target")))`	-

Usage

You can set these function attributes in the declaration, the definition, or both. For example:

```
void AddGlobals(void) __attribute__((always_inline));
__attribute__((always_inline)) void AddGlobals(void) {...}
```

When function attributes conflict, the compiler uses the safer or stronger one. For example, `__attribute__((used))` is safer than `__attribute__((unused))`, and `__attribute__((noinline))` is safer than `__attribute__((always_inline))`.

Related references

3.2 __attribute__((always_inline)) function attribute on page 3-76.
3.3 __attribute__((const)) function attribute on page 3-77.
3.4 __attribute__((constructor[(priority)])) function attribute on page 3-78.
3.5 __attribute__((format_arg(string-index))) function attribute on page 3-79.
3.6 __attribute__((malloc)) function attribute on page 3-80.
3.8 __attribute__((nonnull)) function attribute on page 3-82.
3.7 __attribute__((noinline)) function attribute on page 3-81.
3.11 __attribute__((pcs("calling_convention"))) function attribute on page 3-85.
3.12 __attribute__((pure)) function attribute on page 3-86.
3.9 __attribute__((noreturn)) function attribute on page 3-83.
3.10 __attribute__((nothrow)) function attribute on page 3-84.
3.13 __attribute__((section("name"))) function attribute on page 3-87.
3.15 __attribute__((unused)) function attribute on page 3-89.
3.14 __attribute__((used)) function attribute on page 3-88.
3.16 __attribute__((visibility("visibility_type"))) function attribute on page 3-90.
3.17 __attribute__((weak)) function attribute on page 3-91.
3.18 __attribute__((weakref("target"))) function attribute on page 3-92.
2.2 __alignof__ on page 2-64.
2.3 __asm on page 2-66.
2.4 __declspec attributes on page 2-67.

3.2 __attribute__((always_inline)) function attribute

This function attribute indicates that a function must be inlined.

The compiler attempts to inline the function, regardless of the characteristics of the function. However, the compiler does not inline a function if doing so causes problems.

Example

```
static int max(int x, int y) __attribute__((always_inline));
static int max(int x, int y)
{
    return x > y ? x : y; // always inline if possible
}
```

3.3 __attribute__((const)) function attribute

The `const` function attribute specifies that a function examines only its arguments, and has no effect except for the return value. That is, the function does not read or modify any global memory.

If a function is known to operate only on its arguments then it can be subject to common sub-expression elimination and loop optimizations.

This is a much stricter class than `__attribute__((pure))` because functions are not permitted to read global memory.

Example

```
#include <stdio.h>

// __attribute__((const)) functions do not read or modify any global memory
int my_double(int b) __attribute__((const));
int my_double(int b) {
  return b*2;
}

int main(void) {
    int i;
    int result;
    for (i = 0; i < 10; i++)
    {
       result = my_double(i);
       printf (" i = %d ; result = %d \n", i, result);
    }
}
```

3.4 __attribute__((constructor[(priority)])) function attribute

This attribute causes the function it is associated with to be called automatically before `main()` is entered.

Syntax

`__attribute__((constructor[(`*priority*`)]))`

Where *priority* is an optional integer value denoting the priority. A constructor with a low integer value runs before a constructor with a high integer value. A constructor with a priority runs before a constructor without a priority.

Priority values up to and including 100 are reserved for internal use. If you use these values, the compiler gives a warning.

Usage

You can use this attribute for start-up or initialization code. For example, to specify a function that is to be called when a DLL is loaded.

Example

In the following example, the constructor functions are called before execution enters `main()`, in the order specified:

```
void my_constructor1(void) __attribute__((constructor));
void my_constructor2(void) __attribute__((constructor(102)));
void my_constructor3(void) __attribute__((constructor(103)));
void my_constructor1(void) /* This is the 3rd constructor */
{                         /* function to be called */
    printf("Called my_constructor1()\n");
}
void my_constructor2(void) /* This is the 1st constructor */
{                          /* function to be called */
    printf("Called my_constructor2()\n");
}
void my_constructor3(void) /* This is the 2nd constructor */
{                          /* function to be called */
    printf("Called my_constructor3()\n");
}
int main(void)
{
    printf("Called main()\n");
}
```

This example produces the following output:

```
Called my_constructor2()
Called my_constructor3()
Called my_constructor1()
Called main()
```

3.5 __attribute__((format_arg(string-index))) function attribute

This attribute specifies that a function takes a format string as an argument. Format strings can contain typed placeholders that are intended to be passed to `printf`-style functions such as `printf()`, `scanf()`, `strftime()`, or `strfmon()`.

This attribute causes the compiler to perform placeholder type checking on the specified argument when the output of the function is used in calls to a `printf`-style function.

Syntax

`__attribute__((format_arg(`*`string-index`*`)))`

Where *`string-index`* specifies the argument that is the format string argument (starting from one).

Example

The following example declares two functions, `myFormatText1()` and `myFormatText2()`, that provide format strings to `printf()`.

The first function, `myFormatText1()`, does not specify the `format_arg` attribute. The compiler does not check the types of the `printf` arguments for consistency with the format string.

The second function, `myFormatText2()`, specifies the `format_arg` attribute. In the subsequent calls to `printf()`, the compiler checks that the types of the supplied arguments `a` and `b` are consistent with the format string argument to `myFormatText2()`. The compiler produces a warning when a `float` is provided where an `int` is expected.

```
#include <stdio.h>

// Function used by printf. No format type checking.
extern char *myFormatText1 (const char *);

// Function used by printf. Format type checking on argument 1.
extern char *myFormatText2 (const char *) __attribute__((format_arg(1)));


int main(void) {
  int a;
  float b;

  a = 5;
  b = 9.099999;

  printf(myFormatText1("Here is an integer: %d\n"), a); // No type checking. Types match
anyway.
  printf(myFormatText1("Here is an integer: %d\n"), b); // No type checking. Type mismatch,
but no warning

  printf(myFormatText2("Here is an integer: %d\n"), a); // Type checking. Types match.
  printf(myFormatText2("Here is an integer: %d\n"), b); // Type checking. Type mismatch
results in warning
}


$ armclang --target=aarch64-arm-none-eabi -c format_arg_test.c
format_arg_test.c:21:53: warning: format specifies type 'int' but the argument has type
'float' [-Wformat]
  printf(myFormatText2("Here is an integer: %d\n"), b); // Type checking. Type mismatch
results in warning
                                            ~~      ^
                                            %f
1 warning generated.
```

3.6 __attribute__((malloc)) function attribute

This function attribute indicates that the function can be treated like `malloc` and the compiler can perform the associated optimizations.

Example

```
void * foo(int b) __attribute__((malloc));
```

3.7 __attribute__((noinline)) function attribute

This attribute suppresses the inlining of a function at the call points of the function.

`__attribute__((noinline))` can also be applied to constant data, to prevent the compiler from using the value for optimization purposes, without affecting its placement in the object. This is a feature that can be used for patchable constants, that is, data that is later patched to a different value. It is an error to try to use such constants in a context where a constant value is required.

Example

```
/* Prevent y being used for optimization */
const int y = 5 __attribute__((noinline));
/* Suppress inlining of foo() wherever foo() is called */
int foo(void) __attribute__((noinline));
```

3.8 __attribute__((nonnull)) function attribute

This function attribute specifies function parameters that are not supposed to be null pointers. This enables the compiler to generate a warning on encountering such a parameter.

Syntax

`__attribute__((nonnull[(`*arg-index, ...*`)]))`

Where `[(`*arg-index, ...*`)]` denotes an optional argument index list.

If no argument index list is specified, all pointer arguments are marked as nonnull.

Examples

The following declarations are equivalent:

```
void * my_memcpy (void *dest, const void *src, size_t len) __attribute__((nonnull (1, 2)));
```

```
void * my_memcpy (void *dest, const void *src, size_t len) __attribute__((nonnull));
```

3.9 __attribute__((noreturn)) function attribute

This attribute asserts that a function never returns.

Usage

Use this attribute to reduce the cost of calling a function that never returns, such as exit(). If a noreturn function returns to its caller, the behavior is undefined.

Restrictions

The return address is not preserved when calling the `noreturn` function. This limits the ability of a debugger to display the call stack.

3.10 __attribute__((nothrow)) function attribute

This attribute asserts that a call to a function never results in a C++ exception being sent from the callee to the caller.

The ARM library headers automatically add this qualifier to declarations of C functions that, according to the ISO C Standard, can never throw an exception. However, there are some restrictions on the unwinding tables produced for the C library functions that might throw an exception in a C++ context, for example, `bsearch` and `qsort`.

If the compiler knows that a function can never throw an exception, it might be able to generate smaller exception-handling tables for callers of that function.

3.11 __attribute__((pcs("calling_convention"))) function attribute

This function attribute specifies the calling convention on targets with hardware floating-point.

Syntax

`__attribute__((pcs("`*`calling_convention`*`")))`

Where *`calling_convention`* is one of the following:

`aapcs`
: uses integer registers.

`aapcs-vfp`
: uses floating-point registers.

Example

```
double foo (float) __attribute__((pcs("aapcs")));
```

3.12 __attribute__((pure)) function attribute

Many functions have no effects except to return a value, and their return value depends only on the parameters and global variables. Functions of this kind can be subject to data flow analysis and might be eliminated.

Example

```
int bar(int b) __attribute__((pure));
int bar(int b)
{
    return b++;
}
int foo(int b)
{
    int aLocal=0;
    aLocal += bar(b);
    aLocal += bar(b);
    return 0;
}
```

The call to `bar` in this example might be eliminated because its result is not used.

3.13 __attribute__((section("name"))) function attribute

The `section` function attribute enables you to place code in different sections of the image.

Example

In the following example, the function `foo` is placed into an RO section named `new_section` rather than `.text`.

```
int foo(void) __attribute__((section ("new_section")));
int foo(void)
{
  return 2;
}
```

3.14 __attribute__((used)) function attribute

This function attribute informs the compiler that a static function is to be retained in the object file, even if it is unreferenced.

Functions marked with `__attribute__((used))` are tagged in the object file to avoid removal by linker unused section removal.

Note

Static variables can also be marked as used using `__attribute__((used))`.

Example

```
static int lose_this(int);
static int keep_this(int) __attribute__((used));     // retained in object file
static int keep_this_too(int) __attribute__((used)); // retained in object file
```

3.15 __attribute__((unused)) function attribute

The `unused` function attribute prevents the compiler from generating warnings if the function is not referenced. This does not change the behavior of the unused function removal process.

Note

By default, the compiler does not warn about unused functions. Use `-Wunused-function` to enable this warning specifically, or use an encompassing `-W` value such as `-Wall`.

The `__attribute__((unused))` attribute can be useful if you usually want to warn about unused functions, but want to suppress warnings for a specific set of functions.

Example

```
static int unused_no_warning(int b) __attribute__((unused));
static int unused_no_warning(int b)
{
  return b++;
}

static int unused_with_warning(int b);
static int unused_with_warning(int b)
{
  return b++;
}
```

Compiling this example with `-Wall` results in the following warning:

```
armclang --target=aarch64-arm-none-eabi -c test.c -Wall
test2.cpp:10:12: warning: unused function 'unused_with_warning' [-Wunused-function]
static int unused_with_warning(int b)
           ^
1 warning generated.
```

Related references

3.30 __attribute__((unused)) variable attribute on page 3-104.

3.16 __attribute__((visibility("visibility_type"))) function attribute

This function attribute affects the visibility of ELF symbols.

Syntax

`__attribute__((visibility("`*visibility_type*`")))`

Where *visibility_type* is one of the following:

`default`
: The assumed visibility of symbols can be changed by other options. Default visibility overrides such changes. Default visibility corresponds to external linkage.

`hidden`
: The symbol is not placed into the dynamic symbol table, so no other executable or shared library can directly reference it. Indirect references are possible using function pointers.

`protected`
: The symbol is placed into the dynamic symbol table, but references within the defining module bind to the local symbol. That is, the symbol cannot be overridden by another module.

Usage

Except when specifying `default` visibility, this attribute is intended for use with declarations that would otherwise have external linkage.

You can apply this attribute to functions and variables in C and C++. In C++, it can also be applied to class, struct, union, and enum types, and namespace declarations.

In the case of namespace declarations, the visibility attribute applies to all function and variable definitions.

Example

```
void __attribute__((visibility("protected"))) foo()
{
   ...
}
```

3.17 __attribute__((weak)) function attribute

Functions defined with `__attribute__((weak))` export their symbols weakly.

Functions declared with `__attribute__((weak))` and then defined without `__attribute__((weak))` behave as *weak* functions.

Example

```
extern int Function_Attributes_weak_0 (int b) __attribute__((weak));
```

3.18 __attribute__((weakref("target"))) function attribute

This function attribute marks a function declaration as an alias that does not by itself require a function definition to be given for the target symbol.

Syntax

```
__attribute__((weakref("target")))
```

Where *target* is the target symbol.

Example

In the following example, `foo()` calls `y()` through a weak reference:

```
extern void y(void);
static void x(void) __attribute__((weakref("y")));
void foo (void)
{
  ...
  x();
  ...
}
```

Restrictions

This attribute can only be used on functions with static linkage.

3.19 Type attributes

The `__attribute__` keyword enables you to specify special attributes of variables or structure fields, functions, and types.

The keyword format is either of the following:

```
__attribute__((attribute1, attribute2, ...))
__attribute__((__attribute1__, __attribute2__, ...))
```

For example:

```
typedef union { int i; float f; } U __attribute__((transparent_union));
```

The available type attributes are as follows:

- `__attribute__((aligned))`
- `__attribute__((packed))`
- `__attribute__((transparent_union))`

Related references

3.20 __attribute__((aligned)) type attribute on page 3-94.
3.22 __attribute__((transparent_union)) type attribute on page 3-96.
3.21 __attribute__((packed)) type attribute on page 3-95.

3.20 __attribute__((aligned)) type attribute

The `aligned` type attribute specifies a minimum alignment for the type.

3.21 __attribute__((packed)) type attribute

The `packed` type attribute specifies that a type must have the smallest possible alignment.

3.22 __attribute__((transparent_union)) type attribute

The `transparent_union` type attribute enables you to specify a *transparent_union type*.

When a function is defined with a parameter having transparent union type, a call to the function with an argument of any type in the union results in the initialization of a union object whose member has the type of the passed argument and whose value is set to the value of the passed argument.

When a union data type is qualified with `__attribute__((transparent_union))`, the transparent union applies to all function parameters with that type.

Note

Individual function parameters can also be qualified with the corresponding `__attribute__((transparent_union))` variable attribute.

Example

```
typedef union { int i; float f; } U __attribute__((transparent_union));
void foo(U u)
{
    static int s;
    s += u.i;    /* Use the 'int' field */
}
void caller(void)
{
    foo(1);        /* u.i is set to 1 */
    foo(1.0f);     /* u.f is set to 1.0f */
}
```

3.23 Variable attributes

The `__attribute__` keyword enables you to specify special attributes of variables or structure fields, functions, and types.

The keyword format is either of the following:

```
__attribute__((attribute1, attribute2, ...))
__attribute__((__attribute1__, __attribute2__, ...))
```

For example:

```
static int b __attribute__((__unused__));
```

The available variable attributes are as follows:

- `__attribute__((alias))`
- `__attribute__((aligned))`
- `__attribute__((deprecated))`
- `__attribute__((packed))`
- `__attribute__((section("name")))`
- `__attribute__((unused))`
- `__attribute__((used))`
- `__attribute__((weak))`
- `__attribute__((weakref("target")))`

Related references

3.24 __attribute__((alias)) variable attribute on page 3-98.
3.25 __attribute__((aligned)) variable attribute on page 3-99.
3.26 __attribute__((deprecated)) variable attribute on page 3-100.
3.27 __attribute__((packed)) variable attribute on page 3-101.
3.28 __attribute__((section("name"))) variable attribute on page 3-102.
3.30 __attribute__((unused)) variable attribute on page 3-104.
3.29 __attribute__((used)) variable attribute on page 3-103.
3.31 __attribute__((weak)) variable attribute on page 3-105.
3.32 __attribute__((weakref("target"))) variable attribute on page 3-106.

3.24 __attribute__((alias)) variable attribute

This variable attribute enables you to specify multiple aliases for a variable.

Aliases must be defined in the same translation unit as the original variable.

Note

You cannot specify aliases in block scope. The compiler ignores aliasing attributes attached to local variable definitions and treats the variable definition as a normal local definition.

In the output object file, the compiler replaces alias references with a reference to the original variable name, and emits the alias alongside the original name. For example:

```
int oldname = 1;
extern int newname __attribute__((alias("oldname")));
```

This code compiles to:

```
        movw    r0, :lower16:newname
        movt    r0, :upper16:newname
        ldr     r1, [r0]
        ...
        .type   oldname,%object         @ @oldname
        .data
        .globl  oldname
        .align  2
oldname:
        .long   1                       @ 0x1
        .size   oldname, 4
        ...
        .globl  newname
newname = oldname
```

Note

Function names can also be aliased using the corresponding function attribute `__attribute__((alias))`.

Syntax

`type` *`newname`* `__attribute__((alias("`*`oldname`*`")));`

Where:

`oldname`
is the name of the variable to be aliased

`newname`
is the new name of the aliased variable.

Example

```
#include <stdio.h>
int oldname = 1;
extern int newname __attribute__((alias("oldname"))); // declaration
void foo(void)
{
    printf("newname = %d\n", newname); // prints 1
}
```

3.25 __attribute__((aligned)) variable attribute

The `aligned` variable attribute specifies a minimum alignment for the variable or structure field, measured in bytes.

Example

```
/* Aligns on 16-byte boundary */
int x __attribute__((aligned (16)));

/* In this case, the alignment used is the maximum alignment for a scalar data type. For
ARM, this is 8 bytes. */
short my_array[3] __attribute__((aligned));
```

3.26 __attribute__((deprecated)) variable attribute

The `deprecated` variable attribute enables the declaration of a deprecated variable without any warnings or errors being issued by the compiler. However, any access to a `deprecated` variable creates a warning but still compiles.

The warning gives the location where the variable is used and the location where it is defined. This helps you to determine why a particular definition is deprecated.

Example

```
extern int deprecated_var __attribute__((deprecated));
void foo()
{
    deprecated_var=1;
}
```

Compiling this example generates a warning:

```
armclang --target=aarch64-arm-none-eabi -c test_deprecated.c
test_deprecated.c:4:3: warning: 'deprecated_var' is deprecated [-Wdeprecated-declarations]
  deprecated_var=1;
  ^
test_deprecated.c:1:12: note: 'deprecated_var' declared here
  extern int deprecated_var __attribute__((deprecated));
  ^
1 warning generated.
```

3.27 __attribute__((packed)) variable attribute

The `packed` variable attribute specifies that a variable or structure field has the smallest possible alignment. That is, one byte for a variable, and one bit for a field, unless you specify a larger value with the `aligned` attribute.

Example

```
struct
{
    char a;
    int b __attribute__((packed));
} Variable_Attributes_packed_0;
```

Related references

3.25 __attribute__((aligned)) variable attribute on page 3-99.

3.28 __attribute__((section("*name*"))) variable attribute

The `section` attribute specifies that a variable must be placed in a particular data section.

Normally, the ARM compiler places the data it generates in sections like `.data` and `.bss`. However, you might require additional data sections or you might want a variable to appear in a special section, for example, to map to special hardware.

If you use the `section` attribute, read-only variables are placed in RO data sections, writable variables are placed in RW data sections.

If the section name starts with `.bss.`, the variable is placed in a ZI section.

Example

```
/* in RO section */
const int descriptor[3] __attribute__((section ("descr"))) = { 1,2,3 };
/* in RW section */
long long rw_initialized[10] __attribute__((section ("INITIALIZED_RW"))) = {5};
/* in RW section */
long long rw[10] __attribute__((section ("RW")));
/* in ZI section */
int my_zi __attribute__((section (".bss.my_zi_section")));
```

3.29 __attribute__((used)) variable attribute

This variable attribute informs the compiler that a static variable is to be retained in the object file, even if it is unreferenced.

Data marked with `__attribute__((used))` is tagged in the object file to avoid removal by linker unused section removal.

Note

Static functions can also be marked as used using `__attribute__((used))`.

Example

```
static int lose_this = 1;
static int keep_this __attribute__((used)) = 2;     // retained in object file
static int keep_this_too __attribute__((used)) = 3; // retained in object file
```

3.30 __attribute__((unused)) variable attribute

The compiler can warn if a variable is declared but is never referenced. The __attribute__((unused)) attribute informs the compiler that you expect a variable to be unused and tells it not to issue a warning.

Note

By default, the compiler does not warn about unused variables. Use `-Wunused-variable` to enable this warning specifically, or use an encompassing `-W` value such as `-Weverything`.

The `__attribute__((unused))` attribute can be useful if you usually want to warn about unused variables, but want to suppress warnings for a specific set of variables.

Example

```
void foo()
{
    static int aStatic =0;
    int aUnused __attribute__((unused));
    int bUnused;
    aStatic++;
}
```

When compiled with a suitable `-W` setting, the compiler warns that `bUnused` is declared but never referenced, but does not warn about `aUnused`:

```
armclang --target=aarch64-arm-none-eabi -c test_unused.c -Wall
test_unused.c:5:7: warning: unused variable 'bUnused' [-Wunused-variable]
  int bUnused;
      ^
1 warning generated.
```

Related references

3.15 __attribute__((unused)) function attribute on page 3-89.

3.31 __attribute__((weak)) variable attribute

Generates a weak symbol for a variable, rather than the default strong symbol.

```
extern int foo __attribute__((weak));
```

At link time, strong symbols override weak symbols. This lets you replace a weak symbol with a strong symbol by choosing a particular combination of object files to link.

3.32 __attribute__((weakref("*target*"))) variable attribute

This variable attribute marks a variable declaration as an alias that does not by itself require a definition to be given for the target symbol.

Syntax

```
__attribute__((weakref("target")))
```

Where *target* is the target symbol.

Example

In the following example, `a` is assigned the value of `y` through a weak reference:

```
extern int y;
static int x __attribute__((weakref("y")));
void foo (void)
{
  int a = x;
  ...
}
```

Restrictions

This attribute can only be used on variables that are declared as `static`.

Chapter 4
Compiler-specific Pragmas

Summarizes the ARM compiler-specific pragmas that are extensions to the C and C++ Standards.

It contains the following sections:

4.1 #pragma clang system_header

Causes subsequent declarations in the current file to be marked as if they occur in a system header file.

This pragma suppresses the warning messages that the file produces, from the point after which it is declared.

4.2 #pragma once

This pragma enables the compiler to skip subsequent includes of that header file.

`#pragma once` is accepted for compatibility with other compilers, and enables you to use other forms of header guard coding. However, it is preferable to use `#ifndef` and `#define` coding because this is more portable.

Example

The following example shows the placement of a `#ifndef` guard around the body of the file, with a `#define` of the guard variable after the `#ifndef`.

```
#ifndef FILE_H
#define FILE_H
#pragma once         // optional
 ... body of the header file ...
#endif
```

The `#pragma once` is marked as optional in this example. This is because the compiler recognizes the `#ifndef` header guard coding and skips subsequent includes even if `#pragma once` is absent.

4.3 #pragma pack(*n*)

This pragma aligns members of a structure to the minimum of *n* and their natural alignment. Packed objects are read and written using unaligned accesses.

Note

This pragma is a GNU compiler extension that the ARM compiler supports.

Syntax

`#pragma pack(n)`

Where:

n
is the alignment in bytes, valid alignment values being `1`, `2`, `4` and `8`.

Default

The default is `#pragma pack(8)`.

Example

This example demonstrates how `pack(2)` aligns integer variable `b` to a 2-byte boundary.

```
typedef struct
{
    char a;
    int b;
} S;
#pragma pack(2)
typedef struct
{
    char a;
    int b;
} SP;
S var = { 0x11, 0x44444444 };
SP pvar = { 0x11, 0x44444444 };
```

The layout of `S` is:

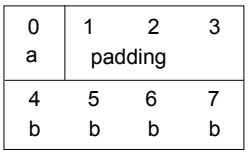

Figure 4-1 Nonpacked structure S

The layout of `SP` is:

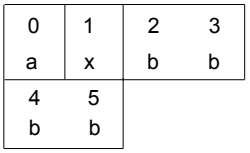

Figure 4-2 Packed structure SP

Note

In this layout, `x` denotes one byte of padding.

`SP` is a 6-byte structure. There is no padding after `b`.

4.4 #pragma unroll[(n)], #pragma unroll_completely

Instructs the compiler to unroll a loop by *n* iterations.

Syntax

```
#pragma unroll

#pragma unroll_completely

#pragma unroll n

#pragma unroll(n)
```

Where:

n
is an optional value indicating the number of iterations to unroll.

Default

If you do not specify a value for *n*, the compiler attempts to fully unroll the loop. The compiler can only fully unroll loops where it can determine the number of iterations.

`#pragma unroll_completely` is a synonym for `#pragma unroll` with no iteration count specified.

Usage

This pragma only has an effect with optimization level `-O2` and higher.

When compiling with `-O3`, the compiler automatically unrolls loops where it is beneficial to do so. You can use this pragma to ask the compiler to unroll a loop that has not been unrolled automatically.

`#pragma unroll[(n)]` can be used immediately before a **`for`** loop, a **`while`** loop, or a **`do`** ... **`while`** loop.

Restrictions

This pragma is a *request* to the compiler to unroll a loop that has not been unrolled automatically. It does not guarantee that the loop is unrolled.

4.5 #pragma weak symbol, #pragma weak symbol1 = symbol2

This pragma is a language extension to mark symbols as weak or to define weak aliases of symbols.

Example

In the following example, `weak_fn` is declared as a weak alias of `__weak_fn`:

```
extern void weak_fn(int a);
#pragma weak weak_fn = __weak_fn
void __weak_fn(int a)
{
    ...
}
```

Chapter 5
Other Compiler-specific Features

Summarizes compiler-specific features that are extensions to the C and C++ Standards, such as predefined macros.

It contains the following sections:

5.1 Predefined macros

The ARM compiler predefines a number of macros. These macros provide information about toolchain version numbers and compiler options.

In general, the predefined macros generated by the compiler are compatible with those generated by GCC. See the GCC documentation for more information.

The following table lists ARM-specific macro names predefined by the ARM compiler for C and C++, together with a number of the most commonly used macro names. Where the value field is empty, the symbol is only defined.

Note

Use `armclang --target=`*`triple`* `-E -dM`*`file`* to see the values of predefined macros.

Table 5-1 Predefined macros

Name	Value	When defined
`__ARM_64BIT_STATE`	`1`	Set for 64-bit targets only. Set to 1 if code is for 64-bit state.
`__ARM_ALIGN_MAX_STACK_PWR`	`4`	Set for 64-bit targets only. The log of the maximum alignment of the stack object.
`__ARM_ARCH`	*`ver`*	Specifies the version of the target architecture, for example 8.
`__ARM_ARCH_EXT_IDIV__`	`1`	Set for 32-bit targets only. Set to 1 if hardware divide instructions are available.
`__ARM_ARCH_ISA_A64`	`1`	Set for 64-bit targets only. Set to 1 if the target supports the A64 instruction set.
`__ARM_ARCH_PROFILE`	*`ver`*	Specifies the profile of the target architecture, for example A.
`__ARM_FEATURE_CLZ`	`1`	Set for 64-bit targets only. Set to 1 if the `CLZ` (count leading zeroes) instruction is supported in hardware.
`__ARM_FEATURE_DIV`	`1`	Set for 64-bit targets only. Set to 1 if the target supports fused floating-point multiply-accumulate.
`__ARM_FEATURE_CRC32`	`1`	Set for 32-bit targets only. Set to 1 if the target has `CRC` extension.
`__ARM_FEATURE_CRYPTO`	`1`	Set to 1 if the target has cryptographic extension.
`__ARM_FEATURE_FMA`	`1`	Set for 64-bit targets only. Set to 1 if the target supports fused floating-point multiply-accumulate.
`__ARM_FEATURE_UNALIGNED`	`1`	Set for 64-bit targets only. Set to 1 if the target unaligned access in hardware.
`__ARM_FP`	`0xE`	Set for 64-bit targets only. Set if hardware floating-point is available.

Table 5-1 Predefined macros (continued)

Name	Value	When defined
__ARM_FP_FAST	1	Set for 64-bit targets only. Set if `-ffast-math` is specified.
__ARM_FP_FENV_ROUNDING	1	Set for 64-bit targets only. Set to 1 if the implementation allows rounding to be configured at runtime using the standard C `fesetround()` function.
__ARM_NEON__	-	Defined when the compiler is targeting an architecture or processor with Advanced SIMD available. Use this macro to conditionally include `arm_neon.h`, to permit the use of Advanced SIMD intrinsics.
__ARM_NEON_FP	7	Set for 64-bit targets only. Set when Advanced SIMD floating-point vector instructions are available.
__ARM_PCS	1	Set for 32-bit targets only. Set to 1 if the default procedure calling standard for the translation unit conforms to the base PCS.
__ARM_PCS_VFP	1	Set for 32-bit targets only. Set to 1 if `-mfloat-abi=hard`.
__ARM_SIZEOF_MINIMAL_ENUM	*value*	Set for 64-bit targets only. Specifies the size of the minimal enumeration type. Set to either 1 or 4 depending on whether `-fshort-enums` is specified or not.
__ARMCOMPILER_VERSION	*nnnbbbb*	Always set. Specifies the version number of the compiler, `armclang`. The format is *nnnbbbb*, where *nnn* is the version number and *bbbb* is the build number. For example, version 6.0 build 0654 is displayed as `6000654`.
__ARMCC_VERSION	*nnnbbbb*	A synonym for `__ARMCOMPILER_VERSION`.
__arm__	1	Defined when targeting the A32 or T32 instruction sets with AArch32 targets, for example `--target=arm-arm-none-eabi`. See also `__aarch64__`.
__aarch64__	1	Defined when targeting the A64 instruction set with `--target=aarch64-arm-none-eabi`. See also `__arm__`.
__cplusplus	*ver*	Defined when compiling C++ code, and set to a value that identifies the targeted C ++ standard. For example, when compiling with `-xc++ -std=gnu++98`, the compiler sets this macro to `199711L`. You can use the `__cplusplus` macro to test whether a file was compiled by a C compiler or a C++ compiler.
__CHAR_UNSIGNED__	1	Defined if and only if **`char`** is an unsigned type.
__EXCEPTIONS	1	Defined when compiling a C++ source file with exceptions enabled.

Table 5-1 Predefined macros (continued)

Name	Value	When defined
__GNUC__	*ver*	Always set. It is an integer that shows the current major version of the compatible GCC version.
__GNUC_MINOR__	*ver*	Always set. It is an integer that shows the current minor version of the compatible GCC version.
__INTMAX_TYPE__	*type*	Always set. Defines the correct underlying type for the `intmax_t` **typedef**.
__NO_INLINE__	1	Defined if no functions have been inlined. The macro is always defined with optimization level `-O0` or if the `-fno-inline` option is specified.
__OPTIMIZE__	1	Defined when `-O1`, `-O2`, `-O3`, `-Ofast`, `-Oz`, or `-Os` is specified.
__OPTIMIZE_SIZE__	1	Defined when `-Os` or `-Oz` is specified.
__PTRDIFF_TYPE__	*type*	Always set. Defines the correct underlying type for the `ptrdiff_t` **typedef**.
__SIZE_TYPE__	*type*	Always set. Defines the correct underlying type for the `size_t` **typedef**.
__SOFTFP__	1	Set for 32-bit targets only. Set to 0 otherwise.
__STDC__	1	Always set. Signifies that the compiler conforms to ISO Standard C.
__STRICT_ANSI__	1	Defined if you specify the `--ansi` option or specify one of the `--std=c*` options .
__thumb__	1	Defined if you specify the `-mthumb` option.
__UINTMAX_TYPE__	*type*	Always set. Defines the correct underlying type for the `uintmax_t` **typedef**.
__VERSION__	*ver*	Always set. A string that shows the underlying Clang version.
__WCHAR_TYPE__	*type*	Always set. Defines the correct underlying type for the `wchar_t` **typedef**.
__WINT_TYPE__	*type*	Always set. Defines the correct underlying type for the `wint_t` **typedef**.

Related references

1.41 --version_number on page 1-56.
1.35 -std on page 1-50.
1.32 -O on page 1-47.
1.37 --target on page 1-52.
1.24 -marm on page 1-38.
1.30 -mthumb on page 1-45.

5.2 Inline functions

Inline functions offer a trade-off between code size and performance. By default, the compiler decides for itself whether to inline code or not.

As a rule, when compiling with `-Ospace`, the compiler makes sensible decisions about inlining with a view to producing code of minimal size. When compiling with `-Otime`, the compiler inlines in most cases, but still avoids large code growth.

In most circumstances, the decision to inline a particular function is best left to the compiler. Qualifying a function with the **`__inline__`** or **`inline`** keywords suggests to the compiler that it inlines that function, but the final decision rests with the compiler. Qualifying a function with **`__attribute((always_inline))`** forces the compiler to inline the function.

The linker is able to apply some degree of function inlining to functions that are very short.

Note

The default semantic rules for C-source code follow C99 rules. For inlining, it means that when you suggest that a function is inlined, the compiler expects to find another, non-qualified, version of the function elsewhere in the code, to use when it decides not to inline. If the compiler cannot find the non-qualified version, it fails with the following error:

`"Error: L6218E: Undefined symbol <symbol> (referred from <file>)"`.

To avoid this problem, there are several options:

- Provide an equivalent, non-qualified version of the function.
- Change the qualifier to **`static inline`**.
- Remove the **`inline`** keyword, since it is only acting as a suggestion.
- Compile your program using the GNU C90 dialect, using the `-std=gnu90` option.

Related references

2.8 __inline on page 2-71.
1.35 -std on page 1-50.

Related information

__attribute((always_inline)).