ARM® Compiler

Version 6.6

armclang Reference Guide

ARM

ARM DUI0774H

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
D	18 November 2015	Non-Confidential	ARM Compiler v6.3 Release
E	24 February 2016	Non-Confidential	ARM Compiler v6.4 Release
F	29 June 2016	Non-Confidential	ARM Compiler v6.5 Release
G	04 November 2016	Non-Confidential	ARM Compiler v6.6 Release
H	08 May 2017	Non-Confidential	ARM Compiler v6.6.1 Release

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* on page 12.

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

This chapter summarizes the supported 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 Intrinsics

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

Chapter 5 Compiler-specific Pragmas

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

Chapter 6 Other Compiler-specific Features

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

Chapter 7 Standard C Implementation Definition

Provides information required by the ISO C standard for conforming C implementations.

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 *ARM Compiler armclang Reference Guide*.
- The number ARM DUI0774H.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

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

Other information

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

Chapter 1
Compiler Command-line Options

This chapter summarizes the supported 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 community feature 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.

It contains the following sections:

1.1 Support level definitions

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

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

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

Identification in the documentation

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

Product features

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

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

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

Beta product features

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

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

Alpha product features

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

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

Community features

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

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

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

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

Guidance on use of community features

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

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

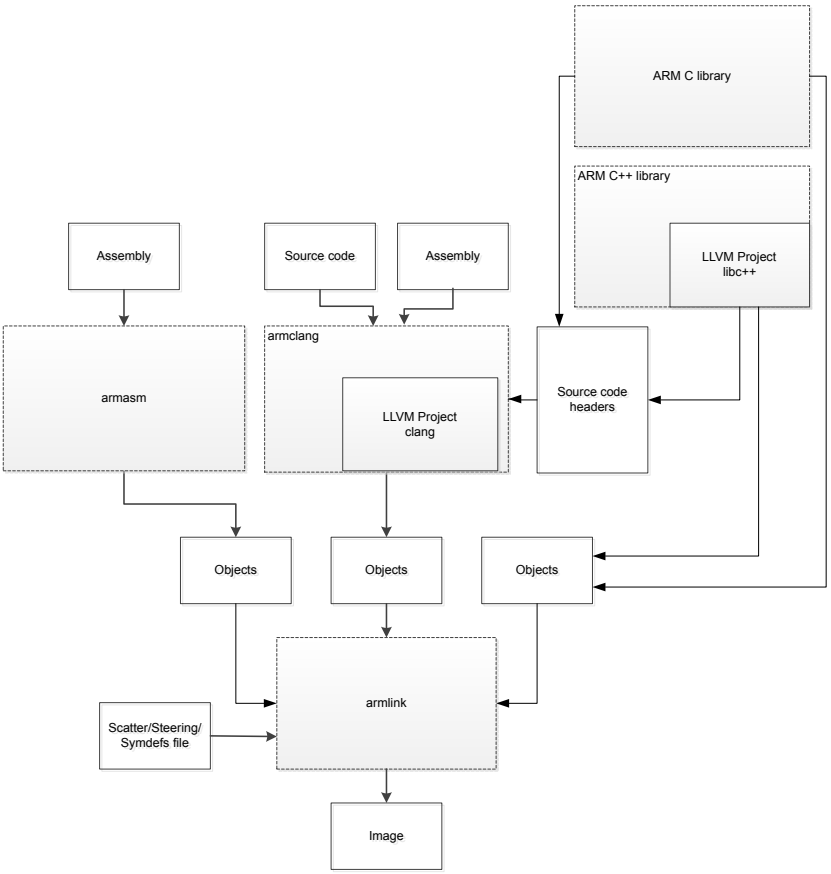

Figure 1-1 Integration boundaries in ARM Compiler 6.

The dashed boxes are toolchain components, and any interaction between these components is an integration boundary. Community features that span an integration boundary might have significant limitations in functionality. The exception to this is if the interaction is codified in one of the standards supported by ARM Compiler 6. See *Application Binary Interface (ABI) for the ARM®*

Architecture. Community features that do not span integration boundaries are more likely to work as expected.
- Features primarily used when targeting hosted environments such as Linux or BSD, might have significant limitations, or might not be applicable, when targeting bare-metal environments.
- The Clang implementations of compiler features, particularly those that have been present for a long time in other toolchains, are likely to be mature. The functionality of new features, such as support for new language features, is likely to be less mature and therefore more likely to have limited functionality.

Unsupported features

With both the product and community feature categories, specific features and use-cases are known not to function correctly, or are not intended for use with ARM Compiler 6.

Limitations of product features are stated in the documentation. ARM cannot provide an exhaustive list of unsupported features or use-cases for community features. The known limitations on community features are listed in *Community features* on page 1-17.

List of known unsupported features

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

  Note

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

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

1.2 -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 -mcpu=cortex-a53 -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 -mcpu=cortex-a53 -c test1.c test2.c -o test.o
armclang: error: cannot specify -o when generating multiple output files
```

1.3 -D

Defines a 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.

Related references

1.31 -include on page 1-51.
1.62 -U on page 1-88.
1.71 -x on page 1-97.

Related information

Preprocessing assembly code.

1.4 -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 also 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.

Examples

Use `-E -dD` to generate interleaved macro definitions and preprocessor output:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -E -dD source.c > raw.c
```

Use `-E -dM` to list all the macros that are defined at the end of the translation unit, including the predefined macros:

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-m3 -E -dM source.c
```

Related references

1.61 --target on page 1-87.

1.5 -e

Specifies the unique initial entry point of the image.

If linking, `armclang` translates this option to `--entry` and passes it to `armlink`. If the link step is not being performed, this option is ignored.

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

Related information

armlink User Guide.

1.6 -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

- This option is unsupported for AArch64 state.
- Bare-metal PIE support is deprecated in this release.

Related references

1.19 -fropi, -fno-ropi on page 1-38.

1.21 -frwpi, -fno-rwpi on page 1-40.

Related information

Bare-metal Position Independent Executables.

--fpic armlink option.

--pie armlink option.

--bare_metal_pie armlink option.

--ref_pre_init armlink option.

1.7 -fbracket-depth=N

Sets the limit for nested parentheses, brackets, and braces to *N* in blocks, declarators, expressions, and struct or union declarations.

Syntax

`-fbracket-depth=N`

Usage

You can increase the depth limit *N*.

Default

The default depth limit is 256.

Related references

7.3 Translation limits on page 7-192.

1.8 -fcommon, -fno-common

Generates common zero-initialized values for tentative definitions.

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

The `-fcommon` option places the tentative 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.

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 of the same symbol in different files can cause a `L6200E: Symbol multiply defined` linker error, because the individual definitions clash with each other.

Default

The default is `-fno-common`.

1.9 -fdata-sections, -fno-data-sections

Enables or disables the generation of one ELF section for each variable in the source file. The default is `-fdata-sections`.

Note

If you want to place specific data items or structures in separate sections, mark them individually with `__attribute__((section("name")))`.

Example

```
volatile int a = 9;
volatile int c = 10;
volatile int d = 11;

int main(void){
    static volatile int b = 2;
    return a == b;
}
```

Compile this code with:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -fdata-sections -c -O3 main.c
```

Use fromelf to see the data sections:

```
fromelf -cds main.o
```

```
...
    Symbol table .symtab (17 symbols, 11 local)

      #  Symbol Name                Value      Bind  Sec  Type  Vis  Size
    ============================================================================

     10  .L_MergedGlobals           0x00000000   Lc   10  Data  De   0x8
     11  main.b                     0x00000004   Lc   10  Data  De   0x4
     12  ...
     13  ...
     14  a                          0x00000000   Gb   10  Data  De   0x4
     15  c                          0x00000000   Gb    7  Data  Hi   0x4
     16  d                          0x00000000   Gb    8  Data  Hi   0x4
...
```

If you compile this code with `-fno-data-sections`, you get:

```
    Symbol table .symtab (15 symbols, 10 local)

      #  Symbol Name                Value      Bind  Sec  Type  Vis  Size
    ============================================================================

      8  .L_MergedGlobals           0x00000008   Lc    7  Data  De   0x8
      9  main.b                     0x0000000c   Lc    7  Data  De   0x4
     10  ...
     11  ...
     12  a                          0x00000008   Gb    7  Data  De   0x4
     13  c                          0x00000000   Gb    7  Data  Hi   0x4
     14  d                          0x00000004   Gb    7  Data  Hi   0x4
...
```

If you compare the two Sec columns, you can see that when `-fdata-sections` is used, the variables are put into different sections. When `-fno-data-sections` is used, all the variables are put into the same section.

Related references

1.12 -ffunction-sections, -fno-function-sections on page 1-31.
3.33 __attribute__((section("name"))) variable attribute on page 3-146.

1.10 -ffast-math, -fno-fast-math

`-ffast-math` tells the compiler to perform more aggressive floating-point optimizations.

`-ffast-math` 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. ARM recommends that you use the alias option `-ffp-mode=fast` instead of `-ffast-math`.

Using `-fno-fast-math` disables aggressive floating-point optimizations. It also ensures that the floating-point code that the compiler generates is compliant with the IEEE Standard for Floating-Point Arithmetic (IEEE 754). ARM recommends that you use the alias option `-ffp-mode=full` instead of `-fno-fast-math`

Note

ARM Compiler 6 uses neither `-ffast-math` nor `-fno-fast-math` by default. For the default behavior, ensure you specify neither `-ffast-math` nor `-fno-fast-math`.

These options control which floating-point library the compiler uses. For more information, see the *library variants* in *ARM C and C++ Libraries and Floating-Point Support User Guide*.

Table 1-1 Floating-point library variants

armclang option	Floating-point library variant	Description
Default	fz	IEEE-compliant library with fixed rounding mode and support for certain IEEE exceptions, and flushing to zero.
-ffast-math	fz	Similar to the default behavior, but also performs aggressive floating-point optimizations and therefore it is not IEEE-compliant.
-fno-fast-math	g	IEEE-compliant library with configurable rounding mode and support for all IEEE exceptions, and flushing to zero.

1.11 -ffp-mode

`-ffp-mode` specifies floating-point standard conformance. This controls which floating-point optimizations the compiler can perform, and also influences library selection.

Syntax

`-ffp-mode=model`

Where *model* is one of the following:

`std`

IEEE finite values with denormals flushed to zero, round-to-nearest, and no exceptions. This is compatible with standard C and C++ and is the default option.
Normal finite values are as predicted by the IEEE standard. However:

- NaNs and infinities might not be produced in all circumstances defined by the IEEE model. When they are produced, they might not have the same sign.
- The sign of zero might not be that predicted by the IEEE model.
- Using NaNs in arithmetic operations with `-ffp-mode=std` causes undefined behavior.

`fast`

Perform more aggressive floating-point optimizations that might cause a small loss of accuracy to provide a significant performance increase. This option defines the symbol `__ARM_FP_FAST`.

This option 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.

A number of transformations might be performed, including:

- Double-precision floating-point expressions that are narrowed to single-precision are evaluated in single-precision when it is beneficial to do so. For example, `float y = (float)(x + 1.0)` is evaluated as `float y = (float)x + 1.0f`.
- Division by a floating-point constant is replaced by multiplication with its reciprocal. For example, `x / 3.0` is evaluated as `x * (1.0 / 3.0)`.
- It is not guaranteed that the value of `errno` is compliant with the ISO C or C++ standard after math functions have been called. This enables the compiler to inline the VFP square root instructions in place of calls to `sqrt()` or `sqrtf()`.

Using a NaN with `-ffp-mode=fast` can produce undefined behavior.

`full`

All facilities, operations, and representations guaranteed by the IEEE Standard for Floating-Point Arithmetic (IEEE 754) are available in single and double-precision. Modes of operation can be selected dynamically at runtime.

These options control which floating-point library the compiler uses. For more information, see the *library variants* in *ARM C and C++ Libraries and Floating-Point Support User Guide*.

Table 1-2 Floating-point library variant selection

armclang option	Floating-point library variant	Description
-ffp-mode=std	fz	IEEE-compliant library with fixed rounding mode and support for certain IEEE exceptions, and flushing to zero.
-ffp-mode=fast	fz	Similar to the default behavior, but also performs aggressive floating-point optimizations and therefore it is not IEEE-compliant.
-ffp-mode=full	g	IEEE-compliant library with configurable rounding mode and support for all IEEE exceptions, and flushing to zero.

Default

The default is `-ffp-mode=std`.

1.12 -ffunction-sections, -fno-function-sections

`-ffunction-sections` generates a separate ELF section for each function in the source file.

`-ffunction-sections` is set by default. The output section for each function has the same name as the function that generates the section, but with a `.text.` prefix. To disable this, use `-fno-function-sections`.

Note

If you want to place specific data items or structures in separate sections, mark them individually with `__attribute__((section("*name*")))`.

Restrictions

`-ffunction-sections` reduces the potential for sharing addresses, data, and string literals between functions. Consequently, it might increase code size slightly for some functions.

Example

```
int function1(int x)
{
  return x+1;
}

int function2(int x)
{
  return x+2;
}
```

Compiling this code with `-ffunction-sections` produces:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -ffunction-sections -S -O3 -o- main.c

...
    .section    .text.function1,"ax",%progbits
    .globl      function1
    .p2align    2
    .type       function1,%function
function1:                              @ @function1
    .fnstart
@ BB#0:
    add         r0, r0, #1
    bx          lr
.Lfunc_end0:
    .size      function1, .Lfunc_end0-function1
    .cantunwind
    .fnend

    .section    .text.function2,"ax",%progbits
    .globl     function2
    .p2align    2
    .type      function2,%function
function2:                              @ @function2
    .fnstart
@ BB#0:
    add        r0, r0, #2
    bx         lr
.Lfunc_end1:
    .size       function2, .Lfunc_end1-function2
    .cantunwind
    .fnend
...
```

Related references

3.17 __attribute__((section("name"))) function attribute on page 3-130.
1.9 -fdata-sections, -fno-data-sections on page 1-27.

1.13 @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.

Note

To use Windows-style file paths on the command-line, you must escape the backslashes. For example:

```
-I"..\\my libs\\".
```

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
```

1.14 -fldm-stm, -fno-ldm-stm

Enable or disable the generation of `LDM` and `STM` instructions. AArch32 only.

Usage

The `-fno-ldm-stm` option can reduce interrupt latency on ARM systems that:

- Do not have a cache or a write buffer.
- Use zero-wait-state, 32-bit memory.

Note

Using `-fno-ldm-stm` might slightly increase code size and decrease performance.

Restrictions

Existing `LDM` and `STM` instructions (for example, in assembly code you are assembling with `armclang`) are not removed.

Default

The default is `-fldm-stm`. That is, by default `armclang` can generate `LDM` and `STM` instructions.

1.15 -fno-inline-functions

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

The compiler attempts to automatically inline functions at optimization levels `-O2` and `-O3`. When these levels are used with `-fno-inline-functions`, automatic inlining is disabled.

When optimization levels `-O0` and `-O1` are used with `-fno-inline-functions`, no automatic inlining is attempted, and only functions that are tagged with `__attribute__((always_inline))` are inlined.

Related concepts

6.3 Inline functions on page 6-183.

Related references

1.55 -O on page 1-80.

1.16 -flto

Enables link time optimization, and outputs bitcode wrapped in an ELF file for link time optimization.

The primary use for files containing bitcode 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.

Usage

The compiler creates one 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.

`-flto` is automatically enabled when you specify the `armclang -Omax` option.

Note

Object files produced with `-flto` contain bitcode, which cannot be disassembled into meaningful disassembly using the `-S` option or the `fromelf` tool.

Caution

Object files generated using the `-flto` option are not suitable for creating static libraries, or ROPI or RWPI images.

Caution

Link Time Optimization performs aggressive optimizations. Sometimes this can result in large chunks of code being removed.

Note

Link Time Optimization does not honor the `armclang -mexecute-only` option. If you use the `armclang -flto` or `-Omax` options, then the compiler cannot generate execute-only code.

Related references

1.2 -c on page 1-20.

Related information

Optimizing across modules with link time optimization.

Restrictions with link time optimization.

--lto armlink option.

1.17 -fexceptions, -fno-exceptions

Enables or disables the generation of code needed to support C++ exceptions.

Default

The default is `-fexceptions` for C++ sources. The default is `-fno-exceptions` for C sources.

Usage

Compiling with `-fno-exceptions` disables exceptions support and uses the variant of C++ libraries without exceptions. Use of try, catch, or throw results in an error message.

Linking objects that have been compiled with `-fno-exceptions` automatically selects the libraries without exceptions. You can use the linker option `--no_exceptions` to diagnose whether the objects being linked contain exceptions.

Note

If an exception propagates into a function that has been compiled without exceptions support, then the program terminates.

Related information

Standard C++ library implementation definition.

1.18 -fomit-frame-pointer, -fno-omit-frame-pointer

`-fomit-frame-pointer` omits the storing of stack frame pointers during function calls.

The `-fomit-frame-pointer` option instructs the compiler to not store stack frame pointers if the function does not need it. You can use this option to reduce the code image size.

The `-fno-omit-frame-pointer` option instructs the compiler to store the stack frame pointer in a register. In AArch32, the frame pointer is stored in register `R11` for ARM code or register `R7` for Thumb code. In AArch64, the frame pointer is stored in register `X29`. The register that is used as a frame pointer is not available for use as a general-purpose register. It is available as a general-purpose register if you compile with `-fomit-frame-pointer`.

Frame pointer limitations for stack unwinding

Frame pointers enable the compiler to insert code to remove the automatic variables from the stack when C++ exceptions are thrown. This is called stack unwinding. However, there are limitations on how the frame pointers are used:

- If you specify `-fomit-frame-pointer`, which is enabled by default, then there are no guarantees on the use of the frame-pointers.
- There are no guarantees about the use of frame pointers in the C or C++ libraries.
- If you specify `-fno-omit-frame-pointer`, then any function which uses space on the stack creates a frame record, and changes the frame pointer to point to it. There is a short time period at the beginning and end of a function where the frame pointer points to the frame record in the caller's frame.
- If you specify `-fno-omit-frame-pointer`, then the frame pointer always points to the lowest address of a valid frame record. A frame record consists of two words:
  - the value of the frame pointer at function entry in the lower-addressed word.
  - the value of the link register at function entry in the higher-addressed word.
- A function that does not use any stack space does not need to create a frame record, and leaves the frame pointer pointing to the caller's frame.
- In AArch32 state, there is currently no reliable way to unwind mixed ARM and Thumb code using frame pointers.
- The behavior of frame pointers in AArch32 state is not part of the ABI and therefore might change in the future. The behavior of frame pointers in AArch64 state is part of the ABI and is therefore unlikely to change.

Default

The default is `-fomit-frame-pointer`.

1.19 -fropi, -fno-ropi

Enables or disables the generation of Read-Only Position-Independent (ROPI) code.

Usage

When generating ROPI code, the compiler:

- Addresses read-only code and data PC-relative.
- Sets the Position Independent (PI) attribute on read-only output sections.

Note

- This option is independent from `-frwpi`, meaning that these two options can be used individually or together.
- When using `-fropi`, `-fropi-lowering` is automatically enabled.

Default

The default is `-fno-ropi`.

Restrictions

The following restrictions apply:

- This option is not supported in AArch64 mode.
- This option cannot be used with C++ code.
- This option is not compatible with `-fpic`, `-fpie`, or `-fbare-metal-pie` options.

Related references

1.21 -frwpi, -fno-rwpi on page 1-40.
1.22 -frwpi-lowering, -fno-rwpi-lowering on page 1-41.
1.20 -fropi-lowering, -fno-ropi-lowering on page 1-39.

1.20 -fropi-lowering, -fno-ropi-lowering

Enables and disables runtime static initialization when generating Read-Only Position-Independent (ROPI) code.

If you compile with `-fropi-lowering`, then the static initialization is done at runtime. It is done by the same mechanism that is used to call the constructors of static C++ objects that must run before `main()`. This enables these static initializations to work with ROPI code.

Default

The default is `-fno-ropi-lowering`. If `-fropi` is used, then the default is `-fropi-lowering`. If `-frwpi` is used without `-fropi`, then the default is `-fropi-lowering`.

1.21 -frwpi, -fno-rwpi

Enables or disables the generation of Read/Write Position-Independent (RWPI) code.

Usage

When generating RWPI code, the compiler:

- Addresses the writable data using offsets from the static base register `sb`. This means that:
  - The base address of the RW data region can be fixed at runtime.
  - Data can have multiple instances.
  - Data can be, but does not have to be, position-independent.
- Sets the PI attribute on read/write output sections.

Note

- This option is independent from `-fropi`, meaning that these two options can be used individually or together.
- When using `-frwpi`, `-frwpi-lowering` and `-fropi-lowering` are automatically enabled.

Restrictions

The following restrictions apply:

- This option is not supported in AArch64 mode.
- This option is not compatible with `-fpic`, `-fpie`, or `-fbare-metal-pie` options.

Default

The default is `-fno-rwpi`.

Related references

1.19 -fropi, -fno-ropi on page 1-38.
1.20 -fropi-lowering, -fno-ropi-lowering on page 1-39.
1.22 -frwpi-lowering, -fno-rwpi-lowering on page 1-41.

1.22 -frwpi-lowering, -fno-rwpi-lowering

Enables and disables runtime static initialization when generating Read-Write Position-Independent (RWPI) code.

If you compile with `-frwpi-lowering`, then the static initialization is done at runtime by the C++ constructor mechanism for both C and C++ code. This enables these static initializations to work with RWPI code.

Default

The default is `-fno-rwpi-lowering`. If `-frwpi` is used, then the default is `-frwpi-lowering`.

1.23 -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* states that the size of enumeration types must be at least 32 bits.

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-none-eabi -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-none-eabi -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).

1.24 -fshort-wchar, -fno-short-wchar

`-fshort-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 -mcpu=cortex-a53 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 -mcpu=cortex-a53 wchar_test.c

size of wchar_t is 2
```

1.25 -fstrict-aliasing, -fno-strict-aliasing

Instructs the compiler to apply the strictest aliasing rules available.

Usage

`-fstrict-aliasing` is implicitly enabled at `-O1` or higher. It is disabled at `-O0`, or when no optimization level is specified.

When optimizing at `-O1` or higher, this option can be disabled with `-fno-strict-aliasing`.

Note

Specifying `-fstrict-aliasing` on the command-line has no effect, since it is either implicitly enabled, or automatically disabled, depending on the optimization level that is used.

Examples

In the following example, `-fstrict-aliasing` is enabled:

```
armclang --target=aarch64-arm-none-eabi -O2 -c hello.c
```

In the following example, `-fstrict-aliasing` is disabled:

```
armclang --target=aarch64-arm-none-eabi -O2 -fno-strict-aliasing -c hello.c
```

In the following example, `-fstrict-aliasing` is disabled:

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

1.26 -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.

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. When using `-fvectorize` with `-O1`, vectorization might be inhibited in the absence of other optimizations which might be present at `-O2` or higher.

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

Using `-fno-vectorize` does not necessarily prevent the compiler from emitting Advanced SIMD instructions. The compiler or linker might still introduce Advanced SIMD instructions, such as when linking libraries that contain these instructions.

Examples

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

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

To prevent the compiler from emitting Advanced SIMD instructions for AArch64 targets, specify `+nosimd` using `-march` or `-mcpu`. For example:

```
armclang --target=aarch64-arm-none-eabi -march=armv8-a+nosimd -O2 file.c -c -S -o file.s
```

To prevent the compiler from emitting Advanced SIMD instructions for AArch32 targets, set the option `-mfpu` to the correct value that does not include Advanced SIMD, for example `fp-armv8`:

```
armclang --target=aarch32-arm-none-eabi -march=armv8-a -mfpu=fp-armv8 -O2 file.c -c -S -o
file.s
```

Related references

1.2 -c on page 1-20.
1.55 -O on page 1-80.

1.27 -ftrapv

Instructs the compiler to generate traps for signed arithmetic overflow on addition, subtraction, and multiplication operations.

Usage

Where an overflow is detected, an undefined instruction is inserted into the assembly code. In order for the overflow to get caught, an undefined instruction handler must be provided.

Note

When both `-fwrapv` and `-ftrapv` are used in a single command, the furthest-right option overrides the other.

For example, here `-ftrapv` overrides `-fwrapv`:

```
armclang --target=aarch64-arm-none-eabi -fwrapv -c -ftrapv hello.c
```

1.28 -fwrapv

Instructs the compiler to assume that signed arithmetic overflow of addition, subtraction, and multiplication, wraps using two's-complement representation.

Note

When both `-fwrapv` and `-ftrapv` are used in a single command, the furthest-right option overrides the other.

For example, here `-fwrapv` overrides `-ftrapv`:

```
armclang --target=aarch64-arm-none-eabi -ftrapv -c -fwrapv hello.c
```

1.29 -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. When using `-g`, the default level is DWARF 4.

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 the default DWARF level implied by `-g`.
- `-gdwarf-2 -g` produces DWARF 4 debug, because `-g` (a synonym for `-gdwarf-4`) overrides `-gdwarf-2`.

1.30 -I

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

1.31 -include

Includes the source code of the specified file at the beginning of the compilation.

Syntax

`-include` *`filename`*
Where *`filename`* is the name of the file whose source code is to be included.

Note

Any `-D`, `-I`, and `-U` options on the command line are always processed before `-include` *`filename`*.

Related references

1.3 -D on page 1-21.
1.30 -I on page 1-50.
1.62 -U on page 1-88.

1.32 -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 armlink User Guide* 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

armlink User Guide.

1.33 -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

armlink User Guide.

1.34 -M, -MM

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.

`-M` lists both system header files and user header files.

`-MM` lists only user header files.

Note

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

Note

To compile the source files and produce makefile dependency rules, use the `-MD` or `-MMD` option instead of the `-M` or `-MM` option respectively.

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

1.54 -o on page 1-79.
1.35 -MD, -MMD on page 1-55.
1.36 -MF on page 1-56.
1.39 -MT on page 1-59.

1.35 -MD, -MMD

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. Unlike `-M` and `-MM`, that cause compilation to stop after the preprocessing stage, `-MD` and `-MMD` allow for compilation to continue.

`-MD` lists both system header files and user header files.

`-MMD` lists only user header files.

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

1.34 -M, -MM on page 1-54.
1.36 -MF on page 1-56.
1.39 -MT on page 1-59.

1.36 -MF

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

Syntax

`-MF` *filename*

Where:

filename
Specifies the filename for the makefile dependency rules.

Note

The `-MF` option only has an effect when used in conjunction with one of the `-M`, `-MM`, `-MD`, or `-MMD` options.

The `-MF` option overrides the default behavior of sending dependency generation output to the standard output stream, and sends output to the specified filename instead.

`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 -mcpu=cortex-a53 -M source.c
```

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

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M source.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 -mcpu=cortex-a53 -MD source.c
```

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

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -MD source.c -MF deps.mk
```

Related references

1.34 -M, -MM on page 1-54.
1.35 -MD, -MMD on page 1-55.
1.39 -MT on page 1-59.

1.37 -MG

Prints dependency lines for header files even if the header files are missing.

Warning and error messages on missing header files are suppressed, and compilation continues.

Note

The `-MG` option only has an effect when used with one of the following options: `-M` or `-MM`.

Example

`source.c` contains a reference to a missing header file `header.h`:

```
#include <stdio.h>
#include "header.h"

int main(void){
    puts("Hello world\n");
    return 0;
}
```

This first example is compiled without the `-MG` option, and results in an error:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M source.c

source.c:2:10: fatal error: 'header.h' file not found
#include "header.h"
         ^
1 error generated.
```

This second example is compiled with the `-MG` option, and the error is suppressed:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M -MG source.c

source.o: source.c \
  /include/stdio.h \
  header.h
```

1.38 -MP

Emits dummy dependency rules.

These rules work around make errors that are generated if you remove header files without a corresponding update to the makefile.

Note

The `-MP` option only has an effect when used in conjunction with the `-M`, `-MD`, `-MM`, or `-MMD` options.

Examples

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

`source.c` includes a header file:

```
#include <stdio.h>

int main(void){
    puts("Hello world\n");
    return 0;
}
```

This first example is compiled without the `-MP` option, and results in a dependency rule for `source.o`:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M source.c

source.o: source.c \
  /include/stdio.h
```

This second example is compiled with the `-MP` option, and results in a dependency rule for `source.o` and a dummy rule for the header file:

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -M -MP source.c

source.o: source.c \
  /include/stdio.h

/include/stdio.h:
```

1.39 -MT

Changes the target of the makefile dependency rule produced by dependency generating options.

Note

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

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

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 -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 -mcpu=cortex-a53 -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 -mcpu=cortex-a53 -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 -mcpu=cortex-a53 -M test1.c test2.c -MT foo -MT bar
foo bar: test1.c header.h
foo bar: test2.c header.h
```

Related references

1.34 -M, -MM on page 1-54.
1.35 -MD, -MMD on page 1-55.
1.36 -MF on page 1-56.

1.40 -march

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

Syntax

To specify a target architecture, use:

`-march=`*name*

`-march=`*name*`[+[no]`*feature*`+…]` (for architectures with optional extensions)

Where:

name

Specifies the architecture.

To view a list of all the supported architectures, use:

```
-march=list
```

The following are valid `-march` values:

`armv8-a`
: ARMv8 application architecture profile. Valid with both `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`.

`armv8.1-a`
: ARMv8.1 application architecture profile. Valid with both `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`.

`armv8.2-a`
: ARMv8.2 application architecture profile. Valid with both `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`.

`armv8.3-a`
: ARMv8.3 application architecture profile. Valid with both `--target=aarch64-arm-none-eabi` and `--target=arm-arm-none-eabi`.

`armv8-r`
: ARMv8 real-time architecture profile. Only valid with `--target=arm-arm-none-eabi`.

`armv8-m.base`
: ARMv8 baseline microcontroller architecture profile. Derived from the ARMv6-M architecture. Only valid with `--target=arm-arm-none-eabi`.

`armv8-m.main`
: ARMv8 mainline microcontroller architecture profile. Derived from the ARMv7-M architecture. Only valid with `--target=arm-arm-none-eabi`.

`armv7-a`
: ARMv7 application architecture profile. Only valid with `--target=arm-arm-none-eabi`.

`armv7-r`
: ARMv7 real-time architecture profile. Only valid with `--target=arm-arm-none-eabi`.

`armv7-m`
: ARMv7 microcontroller architecture profile. Only valid with `--target=arm-arm-none-eabi`.

`armv7e-m`
: ARMv7 microcontroller architecture profile with DSP extension. Only valid with `--target=arm-arm-none-eabi`.

`armv6-m`
: ARMv6 microcontroller architecture profile. Only valid with `--target=arm-arm-none-eabi`.

feature
: Enables or disables an optional architectural feature. See the documentation for `-mcpu`.

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).

Related references

1.44 -mcpu on page 1-66.
1.41 -marm on page 1-62.
1.51 -mthumb on page 1-76.
1.61 --target on page 1-87.

1.41 -marm

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

Most ARMv7-A (and earlier) processors support two instruction sets: the ARM instruction set, and the Thumb instruction set. ARMv8-A AArch32 continues to support these two instruction sets, but they are renamed as A32 and T32 respectively. ARMv8-A additionally introduces the A64 instruction set, used in the AArch64 execution state.

Different architectures support different instruction sets:

- ARMv8-A processors in AArch64 state execute A64 instructions.
- ARMv8-A processors in AArch32 state, as well as ARMv7 and earlier A- and R- profile processors execute A32 (formerly ARM) and T32 (formerly Thumb) instructions.
- M-profile processors execute T32 (formerly Thumb) instructions.

The `-marm` option targets the A32 (formerly ARM) instruction set.

Note

This option is only valid for targets that support the A32 or ARM instruction sets. For example, the `-marm` option is not valid with AArch64 targets. The compiler ignores the `-marm` option and generates a warning with AArch64 targets.

Default

The default for all targets that support ARM or A32 instructions is `-marm`.

Related references

1.51 -mthumb on page 1-76.

1.61 --target on page 1-87.

1.44 -mcpu on page 1-66.

Related information

Specifying a target architecture, processor, and instruction set.

1.42 -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

1.49 -mlittle-endian on page 1-74.

1.43 -mcmse

Enables the generation of code for the Secure state of the ARMv8-M Security Extensions. This option is required when creating a Secure image.

Note

ARMv8-M Security Extensions are not supported when building *Read-Only Position-Independent* (ROPI) and *Read-Write Position-Independent* (RWPI) images.

Usage

Specifying `-mcmse` targets the Secure state of the ARMv8-M Security Extensions. The following are available:

- The Test Target, `TT`, instruction.
- `TT` instruction intrinsics.
- Non-secure function pointer intrinsics.
- `__attribute__((cmse_nonsecure_call))` and `__attribute__((cmse_nonsecure_entry))` function attributes.

Note

- The value of the `__ARM_FEATURE_CMSE` predefined macro indicates what ARMv8-M Security Extension features are supported.
- Compile Secure code with the maximum capabilities for the target. For example, if you compile with no FPU then the Secure functions do not clear floating-point registers when returning from functions declared as `__attribute__((cmse_nonsecure_entry))`. Therefore, the functions could potentially leak sensitive data.
- Structs with undefined bits caused by padding and half float are currently unsupported as arguments and return values for Secure functions. Using such structs might leak sensitive information. Structs that are large enough to be passed by pointer are also unsupported and produce an error.
- The following cases are not supported when compiling with `-mcmse` and give an error:
  - Variadic entry functions.
  - Entry functions with arguments that do not fit in registers, because there are either many arguments or the arguments have large values.
  - Non-secure function calls with arguments that do not fit in registers, because there are either many arguments or the arguments have large values.

Example

This example shows how to create a Secure image using an input import library, `oldimportlib.o`, and a scatter file, `secure.scat`:

```
armclang --target=arm-arm-none-eabi -march=armv8m.main -mcmse secure.c -o secure.o
armlink secure.o -o secure.axf --import-cmse-lib-out importlib.o --import-cmse-lib-in
oldimportlib.o --scatter secure.scat
```

`armlink` also generates the Secure code import library, `importlib.o` that is required for a Non-secure image to call the Secure image.

Related references

1.40 -march on page 1-60.
1.47 -mfpu on page 1-71.
1.61 --target on page 1-87.
3.3 __attribute__((cmse_nonsecure_call)) function attribute on page 3-116.
3.4 __attribute__((cmse_nonsecure_entry)) function attribute on page 3-117.

6.2 Predefined macros on page 6-178.
6.5 TT instruction intrinsics on page 6-185.
6.6 Non-secure function pointer intrinsics on page 6-188.

Related information

Building Secure and Non-secure Images Using ARMv8-M Security Extensions.
TT, TTT, TTA, TTAT instruction.
--fpu linker option.
--import_cmse_lib_in linker option.
--import_cmse_lib_out linker option.
--scatter linker option.

1.44 -mcpu

Enables code generation for a specific ARM processor.

Syntax

To specify a target processor, use:

`-mcpu=`*`name`*

`-mcpu=`*`name`*`[+[no]`*`feature`*`+…]` (for architectures with optional extensions)

Where:

name

Specifies the processor.

To view a list of all supported processors for your target, use:

`-mcpu=list`

feature

Is an optional architecture feature that might be enabled or disabled by default depending on the architecture or processor.

Note

In general, if an architecture supports the optional feature, then this optional feature is enabled by default. To determine whether the optional feature is enabled, use `fromelf --decode_build_attributes`.

`+feature` enables the feature if it is disabled by default. `+feature` has no effect if the feature is already enabled by default.

`+no`*`feature`* disables the feature if it is enabled by default. `+no`*`feature`* has no effect if the feature is already disabled by default.

Use `+`*`feature`* or `+no`*`feature`* to explicitly enable or disable an optional architecture feature.

For AArch64 targets you can specify one or more of the following features if the architecture supports it:

- `crc` - CRC extension.
- `crypto` - Cryptographic extension.
- `fp` - Floating-point extension.
- `fp16` - ARMv8.2-A half-precision floating-point extension.
- `profile` - ARMv8.2-A statistical profiling extension.
- `ras` - Reliability, Availability, and Serviceability extension.
- `simd` - Advanced SIMD extension.
- `rcpc` - Release Consistent Processor Consistent extension. This extension applies to ARMv8.2 and later Application profile architectures.

For AArch32 targets, you can specify one or more of the following features if the architecture supports it:

- `crc` - CRC extension for architectures ARMv8 and above.
- `dsp` - DSP extension for the ARMv8-M.mainline architecture.
- `fp16` - ARMv8.2-A half-precision floating-point extension.
- `ras` - Reliability, Availability, and Serviceability extension.

Note

For AArch32 targets, you can use `-mfpu` to specify the support for floating-point, Advanced SIMD, and cryptographic extensions.

Note

To write code that generates instructions for these extensions, use the intrinsics described in the *ARM C Language Extensions*.

Usage

You can use `-mcpu` option to enable and disable specific architecture 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.

Note

There is no software floating-point library for AArch64 targets. Therefore, when linking for AArch64 targets, `armlink` uses a library with hardware floating-point and Advanced SIMD code. Though you can prevent floating-point and advanced SIMD code from the compilation block using `-mcpu=`*name*`+nofp +nosimd`, there is no guarantee that the linked image for AArch64 targets is entirely free of floating-point or advanced SIMD code.

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).

To see the default floating-point configuration for your processor:

1. Compile with `-mcpu=`*processor* `-S` to generate the assembler file.
2. Open the assembler file and check that the value for the `.fpu` directive corresponds to one of the `-mfpu` options. No `.fpu` directive implies `-mfpu=none`.

Examples

To list the processors that target the AArch64 state:

```
armclang --target=aarch64-arm-none-eabi -mcpu=list
```

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

1.41 -marm on page 1-62.
1.51 -mthumb on page 1-76.
1.61 --target on page 1-87.
1.47 -mfpu on page 1-71.

Related information

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

1.45 -mexecute-only

Generates execute-only code, and prevents the compiler from generating any data accesses to code sections.

To keep code and data in separate sections, the compiler disables literal pools and branch tables when using the `-mexecute-only` option.

Restrictions

Execute-only code must be Thumb code.

Execute-only code is only supported for:

- Processors that support the ARMv8-M.baseline or ARMv8-M.mainline architecture.
- Processors that support the ARMv7-M architecture, such as the Cortex-M3.

If your application calls library functions, the library objects included in the image are not execute-only compliant. You must ensure these objects are not assigned to an execute-only memory region.

Note

ARM does not provide libraries that are built without literal pools. The libraries still use literal pools, even when you use the `-mexecute-only` option.

Note

Link Time Optimization does not honor the `armclang -mexecute-only` option. If you use the `armclang -flto` or `-Omax` options, then the compiler cannot generate execute-only code.

Related information

Building applications for execute-only memory.

1.46 -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 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

1.47 -mfpu on page 1-71.

1.47 -mfpu

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

Syntax

To view a list of all the supported FPU architectures, use:

`-mfpu=list`

Note

`-mfpu=list` is rejected when targeting AArch64.

Alternatively, to specify a target FPU architecture, use:

`-mfpu=`*name*

Where *name* is one of the following:

`none`
: Prevents the compiler from using hardware-based floating-point functions. If the compiler encounters floating-point types in the source code, it will use software-based floating-point library functions. This is similar to the `-mfloat-abi=soft` option.

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

`vfpv3-d16`
: Enable the ARMv7 VFPv3-D16 floating-point extension. Disable the Advanced SIMD extension.

`vfpv3-fp16`
: Enable the ARMv7 VFPv3 floating-point extension, including the optional half-precision extensions. Disable the Advanced SIMD extension.

`vfpv3-d16-fp16`
: Enable the ARMv7 VFPv3-D16 floating-point extension, including the optional half-precision extensions. Disable the Advanced SIMD extension.

`vfpv3xd`
: Enable the ARMv7 VFPv3XD floating-point extension. Disable the Advanced SIMD extension.

`vfpv3xd-fp16`
: Enable the ARMv7 VFPv3XD floating-point extension, including the optional half-precision extensions. Disable the Advanced SIMD extension.

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

`neon-fp16`
: Enable the ARMv7 VFPv3 floating-point extension, including the optional half-precision extensions, and the Advanced SIMD extension.

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

`vfpv4-d16`
: Enable the ARMv7 VFPv4-D16 floating-point extension. Disable the Advanced SIMD extension.

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

`fpv4-sp-d16`
: Enable the ARMv7 FPv4-SP-D16 floating-point extension.

`fpv5-d16`
: Enable the ARMv7 FPv5-D16 floating-point extension.

`fpv5-sp-d16`
: Enable the ARMv7 FPv5-SP-D16 floating-point 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.
- In ARMv7, the Advanced SIMD extension was called the NEON Advanced SIMD extension.

Note

There is no software floating-point library for AArch64 targets. Therefore, when linking for AArch64 targets, `armlink` uses a library with hardware floating-point and Advanced SIMD code. Though you can prevent floating-point and advanced SIMD code from the compilation block using `-mcpu=`*`name`*`+nofp +nosimd`, there is no guarantee that the linked image for AArch64 targets is entirely free of floating-point or advanced SIMD code.

Default

The default FPU option depends on the target processor.

Related references

1.44 -mcpu on page 1-66.
1.46 -mfloat-abi on page 1-70.
1.61 --target on page 1-87.

Related information

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

1.48 -mimplicit-it

Specifies the behavior of the integrated assembler if there are conditional instructions outside `IT` blocks.

`-mimplicit-it=`*name*

Where *name* is one of the following:

`never`
: In A32 code, the integrated assembler gives a warning when there is a conditional instruction without an enclosing IT block. In T32 code, the integrated assembler gives an error, when there is a conditional instruction without an enclosing IT block.

`always`
: In A32 code, the integrated assembler accepts all conditional instructions without giving an error or warning. In T32 code, the integrated assembler outputs an implicit IT block when there is a conditional instruction without an enclosing IT block. The integrated assembler does not give an error or warning about this.

`arm`
: This is the default. In A32 code, the integrated assembler accepts all conditional instructions without giving an error or warning. In T32 code, the integrated assembler gives an error, when there is a conditional instruction without an enclosing IT block.

`thumb`
: In A32 code, the integrated assembler gives a warning when there is a conditional instruction without an enclosing IT block. In T32 code, the integrated assembler outputs an implicit IT block when there is a conditional instruction without an enclosing IT block. The integrated assembler does not give an error or warning about this in T32 code.

Note

This option has no effect in AArch64 state because the A64 instruction set does not include the `IT` instruction. The integrated assembler gives a warning if you use the `-mimplicit-it` option for A64 code.

Default

The default is `-mimplicit-it=arm`.

1.49 -mlittle-endian

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

Default

The default is `-mlittle-endian`.

Related references

1.42 -mbig-endian on page 1-63.

1.50 -munaligned-access, -mno-unaligned-access

Enables and disables unaligned accesses to data on ARM processors.

The compiler defines the `__ARM_FEATURE_UNALIGNED` macro when `-munaligned-access` is enabled.

The libraries include special versions of certain library functions designed to exploit unaligned accesses. When unaligned access support is enabled, using `-munaligned-access`, the compilation tools use these library functions to take advantage of unaligned accesses. When unaligned access support is disabled, using `-mno-unaligned-access`, these special versions are not used.

Default

`-munaligned-access` is the default for architectures that support unaligned accesses to data. This applies to all architectures supported by ARM Compiler 6, except ARMv6-M and ARMv8-M.baseline.

Usage

`-munaligned-access`
: Use this option on processors that support unaligned accesses to data, to speed up accesses to packed structures.

  Note

  For ARMv6-M and ARMv8-M.baseline architectures, compiling with this option generates an error.

`-mno-unaligned-access`
: If unaligned access is disabled, any unaligned data that is wider than 8-bit is accessed one byte at a time. For example, fields wider than 8-bit, in packed data structures, are always accessed one byte at a time even if they are aligned.

Related references

6.2 Predefined macros on page 6-178.
2.9 __unaligned on page 2-110.

Related information

ARM C Language Extensions 2.0.

1.51 -mthumb

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

Most ARMv7-A (and earlier) processors support two instruction sets: the ARM instruction set, and the Thumb instruction set. ARMv8-A AArch32 continues to support these two instruction sets, but they are renamed as A32 and T32 respectively. ARMv8-A additionally introduces the A64 instruction set, used in the AArch64 execution state.

Different architectures support different instruction sets:

- ARMv8-A processors in AArch64 state execute A64 instructions.
- ARMv8-A processors in AArch32 state, in addition to ARMv7 and earlier A- and R- profile processors execute A32 (formerly ARM) and T32 (formerly Thumb) instructions.
- M-profile processors execute T32 (formerly Thumb) instructions.

The `-mthumb` option targets the T32 (formerly Thumb) 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.
- The `-mthumb` option is recognized when using `armclang` as a compiler, but not when using it as an assembler. To request `armclang` to assemble using the T32 or Thumb instruction set for your assembly source files, you must use the `.thumb` or `.code 16` directive in the assembly files.

Default

The default for all targets that support ARM or A32 instructions is `-marm`.

Example

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

Related references

1.41 -marm on page 1-62.
1.61 --target on page 1-87.
1.44 -mcpu on page 1-66.

Related information

Specifying a target architecture, processor, and instruction set.
Assembling ARM and GNU syntax assembly code.

1.52 -nostdlib

Tells the compiler to not use the ARM standard C and C++ libraries.

If you use the `-nostdlib` option, `armclang` does not collude with the ARM standard library and only emits calls to AEABI compliant library functions. The output from `armclang` works with any ISO C library that is compliant with AEABI.

The `-nostdlib` `armclang` option, passes the `--no_scanlib` linker option to `armlink`. Therefore you must specify the location of the libraries you want to use as input objects to `armlink`, or with the `--userlibpath` `armlink` option.

Note

If you want to use your own libraries instead of the ARM standard libraries or if you want to re-implement the standard library functions, then you must use the `-nostdlib` `armclang` option. Your libraries must be compliant with the ISO C library and with the AEABI specification.

Default

`-nostdlib` is disabled by default.

Example

```
#include "math.h"

double foo(double d)
{
    return sqrt(d + 1.0);
}
int main(int argc, char *argv[])
{
    return foo(argc);
}
```

Compiling this code with `-nostdlib` generates a call to `sqrt` (from the ARM standard library), which is AEABI compliant.

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-A9 -O0 -S -o- file.c -mfloat-abi=hard -
nostdlib
```

Compiling this code without `-nostdlib` generates a call to `__hardfp_sqrt` (from the ARM standard library), which is not AEABI compliant.

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-A9 -O0 -S -o- file.c -mfloat-abi=hard
```

1.53 -nostdlibinc

Tells the compiler to exclude the ARM standard C and C++ library header files.

If you want to disable the use of the ARM standard library, then use the `-nostdlibinc` and `-nostdlib` `armclang` options.

Default

`-nostdlibinc` is disabled by default.

Example

```
#include "math.h"

double foo(double d)
{
    return sqrt(d + 1.0);
}
int main(int argc, char *argv[])
{
    return foo(argc);
}
```

Compiling this code without `-nostdlibinc` generates a call to `__hardfp_sqrt`, from the ARM standard library.

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-A9 -O0 -S -o- file.c -mfloat-abi=hard
```

Compiling this code with `-nostdlibinc` and `-nostdlib` generates an error because the compiler cannot include the standard library header file `math.h`.

```
armclang --target=arm-arm-none-eabi -mcpu=Cortex-A9 -O0 -S -o- file.c -mfloat-abi=hard -
nostdlibinc -nostdlib
```

1.54 -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 1-3 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

1.55 -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 for the performance of the compiled binary. Turns off most optimizations. When debugging is enabled, this option generates code that directly corresponds to the source code. Therefore, this might result in a significantly larger image.

This is the default optimization level.

`1`

Restricted optimization. When debugging is enabled, this option gives the best debug view for the trade-off between image size, performance, and debug.

`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`

Very high 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 level `3` including those performed with the `-ffp-mode=fast` `armclang` option. This level also performs other aggressive optimizations that might violate strict compliance with language standards.

max

Maximum optimization. Specifically targets performance optimization. Enables all the optimizations from level `fast`, together with other aggressive optimizations.

Caution

This option is not guaranteed to be fully standards-compliant for all code cases.

Caution

`-Omax` automatically enables the `armclang -flto` option and the generated object files are not suitable for creating static libraries. When `-flto` is enabled, you cannot build ROPI or RWPI images.

Note

When using `-Omax`:

- Code-size, build-time, and the debug view can each be adversely affected.
- ARM cannot guarantee that the best performance optimization is achieved in all code cases.
- It is not possible to output meaningful disassembly when the `-flto` option is enabled, which is turned on by default at `-Omax`, because this generates files containing bitcode.
- If you are trying to compile at `-Omax` and have separate compile and link steps, then also include `-Omax` on your `armlink` command line.

Note

Link Time Optimization does not honor the `armclang -mexecute-only` option. If you use the `armclang -flto` or `-Omax` options, then the compiler cannot generate execute-only code.

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`. For debug view, ARM recommends `-O1` rather than `-O0` for best trade-off between image size, performance, and debug.

Related references

1.16 -flto on page 1-35.

1.19 -fropi, -fno-ropi on page 1-38.

1.21 -frwpi, -fno-rwpi on page 1-40.

Related information

Restrictions with link time optimization.

1.56 -pedantic

Generate warnings if code violates strict ISO C and ISO C++.

If you use the `-pedantic` option, the compiler generates warnings if your code uses any language feature that conflicts with strict ISO C or ISO C++.

Default

`-pedantic` is disabled by default.

Example

```
void foo(void)
{
    long long i; /* okay in nonstrict C90 */
}
```

Compiling this code with `-pedantic` generates an warning.

```
armclang --target=arm-arm-none-eabi -march=armv8-a file.c -c -std=c90 -pedantic
```

Related references

1.60 -std on page 1-86.
1.68 -W on page 1-94.
1.57 -pedantic-errors on page 1-83.

1.57 -pedantic-errors

Generate errors if code violates strict ISO C and ISO C++.

If you use the `-pedantic-errors` option, the compiler does not use any language feature that conflicts with strict ISO C or ISO C++. The compiler generates an error if your code violates strict ISO language standard.

Default

`-pedantic-errors` is disabled by default.

Example

```
void foo(void)
{
    long long i; /* okay in nonstrict C90 */
}
```

Compiling this code with `-pedantic-errors` generates an error:

```
armclang --target=arm-arm-none-eabi -march=armv8-a file.c -c -std=c90 -pedantic-errors
```

Related references

1.60 -std on page 1-86.
1.68 -W on page 1-94.
1.56 -pedantic on page 1-82.

1.58 -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.

Note

It is not possible to output meaningful disassembly when the `-flto` option is enabled, which is turned on by default at `-Omax`, because this generates files containing bitcode.

Related references

1.54 -o on page 1-79.
1.55 -O on page 1-80.
1.16 -flto on page 1-35.

1.59 -save-temps

Instructs the compiler to generate intermediate assembly files from the specified C/C++ file.

It is similar to disassembling object code that has been compiled from C/C++.

Example

```
armclang --target=aarch64-arm-none-eabi -save-temps -c hello.c
```

Executing this command outputs the following files, that are listed in the order they are created:

- `hello.i` (or `hello.ii` for C++): the C or C++ file after pre-processing.
- `hello.bc`: the llvm-ir bitcode file.
- `hello.s`: the assembly file.
- `hello.o`: the output object file.

Note

- Specifying `-c` means that the compilation process stops after the compilation step, and does not do any linking.
- Specifying `-S` means that the compilation process stops after the disassembly step, and does not create an object file.

Related references

1.2 -c on page 1-20.
1.58 -S on page 1-84.

1.60 -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++03`
C++ as defined by the 2003 standard.

`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.

Note

Use of C11 library features is unsupported.

Related references

1.71 -x on page 1-97.
1.57 -pedantic-errors on page 1-83.
1.56 -pedantic on page 1-82.

Related information

Language Compatibility.
C++ Status.
Language Support Levels.

1.61 --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` or `-march` 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

1.41 -marm on page 1-62.
1.51 -mthumb on page 1-76.
1.44 -mcpu on page 1-66.
1.47 -mfpu on page 1-71.

Related information

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

1.62 -U

Removes any initial definition of the specified macro.

Syntax

`-U`*`name`*

Where:

`name`
is the name of the macro to be undefined.

The macro *`name`* can be either:

- A predefined macro.
- A macro specified using the `-D` option.

Note

Not all compiler predefined macros can be undefined.

Usage

Specifying `-U`*`name`* has the same effect as placing the text `#undef` *`name`* at the head of each source file.

Restrictions

The compiler defines and undefines macros in the following order:

1. Compiler predefined macros.
2. Macros defined explicitly, using `-D`*`name`*.
3. Macros explicitly undefined, using `-U`*`name`*.

Related references

1.3 -D on page 1-21.
6.2 Predefined macros on page 6-178.
1.31 -include on page 1-51.

1.63 -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 armlink User Guide* for information about the `--undefined` linker option.

Related information

armlink User Guide.

1.64 -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

1.72 -### on page 1-98.

1.65 --version

Displays the same information as `--vsn`.

Related references

1.67 --vsn on page 1-93.

1.66 --version_number

Displays the version of `armclang` you are using.

Usage

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

- *`M`* is the major version number, `6`.
- *`mm`* is the minor version number.
- *`uu`* is the update number.
- *`xx`* is reserved for ARM internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions.

Related references

6.2 Predefined macros on page 6-178.

1.67 --vsn on page 1-93.

1.67 --vsn

Displays the version information and the license details.

Note

`--vsn` is intended to report the version information for manual inspection. The `Component` line indicates the release of ARM Compiler you are using. If you need to access the version in other tools or scripts, for example in build scripts, use the output from `--version_number`.

Example

Example output:

```
> armclang --vsnProduct: ARM Compiler N.n.pComponent: ARM Compiler N.n.pTool: armclang
[tool_id]

Target: target_name
```

Related references

1.65 --version on page 1-91.
1.66 --version_number on page 1-92.

1.68 -W

Controls diagnostics.

Syntax

`-W`*name*

Where common values for *name* include:

`-Wc11-extensions`
: Warns about any use of C11-specific features.

`-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.

`-W`*foo*
: Enable warning *foo*.

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

`-Weverything`
: Enable all warnings.

`-Wpedantic`
: Generate warnings if code violates strict ISO C and ISO C++.

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

Related references

1.57 -pedantic-errors on page 1-83.

1.56 -pedantic on page 1-82.

Related information

Options for controlling diagnostics with armclang.

1.69 -Wl

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

See the *ARM® Compiler armlink User Guide* for information about available linker options.

Syntax

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

Where:

opt
is a linker 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 -mcpu=cortex-a53 hello.c -Wl,--split,--list,diag.txt
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 hello.c -Wl,--split,--list=diag.txt
```

Related references

1.70 -Xlinker on page 1-96.

Related information

ARM Compiler Linker Command-line Options.

1.70 -Xlinker

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

See the *ARM® Compiler armlink User Guide* for information about available linker options.

Syntax

`-Xlinker` *opt*

Where:

opt
is a linker 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 -mcpu=cortex-a53 hello.c -Xlinker --split
```

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

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a53 hello.c -Xlinker --list -Xlinker
diag.txt
```

Related references

1.69 -Wl on page 1-95.

Related information

ARM Compiler Linker Command-line Options.

1.71 -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

1.3 -D on page 1-21.

1.60 -std on page 1-86.

Related information

Preprocessing assembly code.

1.72 -###

Displays the commands that invoke the compiler and sub-tools, such as `armlink`, 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

1.64 -v on page 1-90.

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 2-101.
2.3 __asm on page 2-103.
2.4 __declspec attributes on page 2-105.
2.8 __inline on page 2-109.

2.2 __alignof__

The `__alignof__` keyword enables you to inquire 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.

Embedded assembly

For embedded assembly, you cannot use the `__asm` keyword on the function declaration. Use the *`__attribute__((naked))`* on page 3-123 function attribute on the function declaration. For example:

```
__attribute__((naked)) void foo (int i);
```

Naked functions with the `__attribute__((naked))` function attribute only support assembler instructions in the basic format:

`__asm(`*`code`*`);`

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
```

Related references

3.10 __attribute__((naked)) function attribute on page 3-123.

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 2-106.
2.6 __declspec(noreturn) on page 2-107.
2.7 __declspec(nothrow) on page 2-108.

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

6.3 Inline functions on page 6-183.

2.9 __unaligned

The `__unaligned` keyword is a type qualifier that tells the compiler to treat the pointer or variable as an unaligned pointer or variable.

Members of packed structures might be unaligned. Use the `__unaligned` keyword on pointers that you use for accessing packed structures or members of packed structures.

Example

```
typedef struct __attribute__((packed)) S{
    char c;
    int x;
};

int f1_load(__unaligned struct S *s)
{
    return s->x;
}
```

The compiler generates an error if you assign an unaligned pointer to a regular pointer without type casting.

Example

```
struct __attribute__((packed)) S { char c; int x; };
void foo(__unaligned struct S *s2)
{
    int *p = &s2->x;             // compiler error because &s2->x is an unaligned pointer
but p is a regular pointer.
    __unaligned int *q = &s2->x; // No error because q and &s2->x are both unaligned
pointers.
}
```

Related references

1.50 -munaligned-access, -mno-unaligned-access on page 1-75.

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 3-115.
3.5 __attribute__((const)) function attribute on page 3-118.
3.6 __attribute__((constructor[(priority)])) function attribute on page 3-119.
3.7 __attribute__((format_arg(string-index))) function attribute on page 3-120.
3.9 __attribute__((malloc)) function attribute on page 3-122.
3.12 __attribute__((nonnull)) function attribute on page 3-125.
3.10 __attribute__((naked)) function attribute on page 3-123.
3.15 __attribute__((pcs("calling_convention"))) function attribute on page 3-128.
3.11 __attribute__((noinline)) function attribute on page 3-124.
3.14 __attribute__((nothrow)) function attribute on page 3-127.
3.17 __attribute__((section("name"))) function attribute on page 3-130.
3.16 __attribute__((pure)) function attribute on page 3-129.
3.13 __attribute__((noreturn)) function attribute on page 3-126.
3.19 __attribute__((unused)) function attribute on page 3-132.
3.18 __attribute__((used)) function attribute on page 3-131.
3.21 __attribute__((visibility("visibility_type"))) function attribute on page 3-134.
3.22 __attribute__((weak)) function attribute on page 3-135.
3.23 __attribute__((weakref("target"))) function attribute on page 3-136.
2.2 __alignof__ on page 2-101.
2.3 __asm on page 2-103.
2.4 __declspec attributes on page 2-105.

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.

In some circumstances, the compiler might choose to ignore `__attribute__((always_inline))`, and not inline the function. For example:

- A recursive function is never inlined into itself.
- Functions that use `alloca()` might not be inlined.

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__((cmse_nonsecure_call)) function attribute

Declares a non-secure function type

A call to a function that switches state from Secure to Non-secure is called a non-secure function call. A non-secure function call can only happen through function pointers. This is a consequence of separating secure and non-secure code into separate executable files.

A non-secure function type must only be used as a base type of a pointer.

Example

```
#include <arm_cmse.h>
typedef void __attribute__((cmse_nonsecure_call)) nsfunc(void);

void default_callback(void) { … }

// fp can point to a secure function or a non-secure function
nsfunc *fp = (nsfunc *) default_callback;   // secure function pointer

void __attribute__((cmse_nonsecure_entry)) entry(nsfunc *callback) {
    fp = cmse_nsfptr_create(callback);      // non-secure function pointer
}

void call_callback(void) {
    if (cmse_is_nsfptr(fp)){
       fp();           // non-secure function call
    }
    else {
        ((void (*)(void)) fp)();           // normal function call
    }
}
```

Related references

3.4 __attribute__((cmse_nonsecure_entry)) function attribute on page 3-117.

6.6 Non-secure function pointer intrinsics on page 6-188.

Related information

Building Secure and Non-secure Images Using ARMv8-M Security Extensions.

3.4 __attribute__((cmse_nonsecure_entry)) function attribute

Declares an entry function that can be called from Non-secure state or Secure state.

Syntax

C linkage:

```
void __attribute__((cmse_nonsecure_entry)) entry_func(int val)
```

C++ linkage:

```
extern "C" void __attribute__((cmse_nonsecure_entry)) entry_func(int val)
```

Note

Compile Secure code with the maximum capabilities for the target. For example, if you compile with no FPU then the Secure functions do not clear floating-point registers when returning from functions declared as `__attribute__((cmse_nonsecure_entry))`. Therefore, the functions could potentially leak sensitive data.

Example

```
#include <arm_cmse.h>
void __attribute__((cmse_nonsecure_entry)) entry_func(int val) {
  int state = cmse_nonsecure_caller();

  if (state)
  { // called from non-secure
    // do non-secure work
    ...
  } else
  { // called from within secure
    // do secure work
    ...
  }
}
```

Related references

3.3 __attribute__((cmse_nonsecure_call)) function attribute on page 3-116.

6.6 Non-secure function pointer intrinsics on page 6-188.

Related information

Building Secure and Non-secure Images Using ARMv8-M Security Extensions.

3.5 __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 attribute is stricter 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);
    }
}
```

3.6 __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.

Example

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

```
#include <stdio.h>
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()
```

3.7 __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 -mcpu=cortex-a53 -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.
```

3.8 __attribute__((interrupt("type"))) function attribute

The GNU style `interrupt` attribute instructs the compiler to generate a function in a manner that is suitable for use as an exception handler.

Syntax

```
__attribute__((interrupt("type")))
```

Where *type* is one of the following:

- IRQ.
- FIQ.
- SWI.
- ABORT.
- UNDEF.

Usage

The `interrupt` attribute affects the code generation of a function as follows:

- If the function is AAPCS, the stack is realigned to 8 bytes on entry.
- For processors that are not based on the M-profile, preserves all processor registers, rather than only the registers that the AAPCS requires to be preserved. Floating-point registers are not preserved.
- For processors that are not based on the M-profile, the function returns using an instruction that is architecturally defined as a return from exception.

Restrictions

When using `__attribute__((interrupt("type")))` functions:

- No arguments or return values can be used with the functions.
- The functions are incompatible with `-frwpi`.

——— **Note** ———

In ARMv6-M, ARMv7-M, and ARMv8-M, the architectural exception handling mechanism preserves all processor registers, and a standard function return can cause an exception return. Therefore, specifying the `interrupt` attribute does not affect the behavior of the compiled output. However, ARM recommends using the `interrupt` attribute on exception handlers for clarity and easier software porting.

———————————

——— **Note** ———

- For architectures that support A32 and T32 instructions, functions specified with the `interrupt` attribute compile to A32 or T32 code depending on whether the compile option specifies ARM or Thumb.
- For Thumb only architectures, for example ARMv6-M, functions specified with the `interrupt` attribute compile to T32 code.
- The `interrupt` attribute is not available for A64 code.

———————————

3.9 __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));
```

3.10 __attribute__((naked)) function attribute

This attribute tells the compiler that the function is an embedded assembly function. You can write the body of the function entirely in assembly code using `__asm` statements.

The compiler does not generate prologue and epilogue sequences for functions with `__attribute__((naked))`.

The compiler only supports basic `__asm` statements in `__attribute__((naked))` functions. Using extended assembly, parameter references or mixing C code with `__asm` statements might not work reliably.

Example 3-1 Examples

```
__attribute__((naked)) int add(int i, int j); /* Declaring a function with
__attribute__((naked)). */

__attribute__((naked)) int add(int i, int j)
{
    __asm("ADD r0, r1, #1"); /* Basic assembler statements are supported. */

/*  Parameter references are not supported inside naked functions: */
/*  __asm (
    "ADD r0, %[input_i], %[input_j]"       /* Assembler statement with parameter references
*/
    :                                      /* Output operand parameter */
    : [input_i] "r" (i), [input_j] "r" (j) /* Input operand parameter */
    );
*/

/*  Mixing C code is not supported inside naked functions: */
/*  int res = 0;
    return res;
*/

}
```

Related references

2.3 __asm on page 2-103.

3.11 __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));
```

3.12 __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.

Note

The argument index list is 1-based, rather than 0-based.

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));
```

3.13 __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.

3.14 __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.

3.15 __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")));
```

3.16 __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.

3.17 __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;
}
```

Note

Section names must be unique. You must not use the same section name for different section types. If you use the same section name for different section types, then the compiler merges the sections into one and gives the section the type of whichever function or variable is first allocated to that section.

3.18 __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))` can still be removed by linker unused section removal. To prevent the linker from removing these sections use the `--keep armlink` option.

Note

Static variables can also be marked as used, by 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
```

Related information

--keep.

3.19 __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
```

```
test.c:10:12: warning: unused function 'unused_with_warning' [-Wunused-function]
static int unused_with_warning(int b)
           ^
1 warning generated.
```

Related references

3.35 __attribute__((unused)) variable attribute on page 3-148.

3.20 __attribute__((value_in_regs)) function attribute

The `value_in_regs` function attribute is compatible with functions whose return type is a structure. It alters the calling convention of a function so that the returned structure is stored in the argument registers rather than being written to memory using an implicit pointer argument.

——— **Note** ———

When using `__attribute__((value_in_regs))`, the calling convention only uses integer registers.

———————————

Syntax

```
__attribute__((value_in_regs)) return-type function-name([argument-list]);
```

Where:

`return-type`
is the type of structure that conforms to certain restrictions.

Usage

Declaring a function `__attribute__((value_in_regs))` can be useful when calling functions that return more than one result.

Restrictions

When targeting AArch32, the structure can be up to 16 bytes in order to fit in four 32-bit argument registers. When targeting AArch64, the structure can be up to 64 bytes in order to fit in eight 64-bit argument registers. If the structure returned by a function qualified by `__attribute__((value_in_regs))` is too big, the compiler generates an error.

Each field of the structure has to fit exactly in one integer register. Therefore, the fields can only be pointers or pointer-sized integers. Anything else, including bitfields, is incompatible. Nested structures are allowed if they contain a single field whose type is pointer or pointer-sized integer. Unions can have more that one field. If the type of the return structure violates any of the above rules, then the compiler generates the corresponding error.

If a virtual function declared as `__attribute__((value_in_regs))` is to be overridden, the overriding function must also be declared as `__attribute__((value_in_regs))`. If the functions do not match, the compiler generates an error.

A function declared as `__attribute__((value_in_regs))` is not function-pointer-compatible with a normal function of the same type signature. If a pointer to a function that is declared as `__attribute__((value_in_regs))` is initialized with a pointer to a function that is not declared as `__attribute__((value_in_regs))`, then the compiler generates a warning.

Example

```
typedef struct ReturnType
{
    long a;
    char *b;
    union U
    {
       int *c;
       struct S1 {short *d;} s1;
    } u;
    struct S2 {double *e;} s2;
} my_struct;

extern __attribute__((value_in_regs)) my_struct foo(long x);
```

3.21 __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()
{
   ...
}
```

3.22 __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));
```

3.23 __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.

3.24 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

3.25 __attribute__((aligned)) type attribute on page 3-138.
3.27 __attribute__((transparent_union)) type attribute on page 3-140.
3.26 __attribute__((packed)) type attribute on page 3-139.

3.25 __attribute__((aligned)) type attribute

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

3.26 __attribute__((packed)) type attribute

The `packed` type attribute specifies that a type must have the smallest possible alignment. This attribute only applies to `struct` and `union` types.

Note

You must access a `packed` member of a `struct` or `union` directly from a variable of the containing type. Taking the address of such a member produces a normal pointer which might be unaligned. The compiler assumes that the pointer is aligned. Dereferencing such a pointer can be unsafe even when unaligned accesses are supported by the target, because certain instructions always require word-aligned addresses.

Note

If you take the address of a packed member, in most cases, the compiler generates a warning.

When you specify `__attribute__((packed))` to a structure or union, it applies to all members of the structure or union. If a packed structure has a member that is also a structure, then this member structure has an alignment of 1-byte. However, the packed attribute does not apply to the members of the member structure. The members of the member structure continue to have their natural alignment.

Example 3-2 Examples

```
struct __attribute__((packed)) foobar
{
  char x;
  short y;
};

short get_y(struct foobar *s)
{
    // Correct usage: the compiler will not use unaligned accesses
    // unless they are allowed.
    return s->y;
}

short get2_y(struct foobar *s)
{
    short *p = &s->y; // Incorrect usage: 'p' might be an unaligned pointer.
    return *p;  // This might cause an unaligned access.
}
```

Related references

1.50 -munaligned-access, -mno-unaligned-access on page 1-75.

3.27 __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.

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 */
}
```

3.28 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

3.29 __attribute__((alias)) variable attribute on page 3-142.
3.30 __attribute__((aligned)) variable attribute on page 3-143.
3.31 __attribute__((deprecated)) variable attribute on page 3-144.
3.32 __attribute__((packed)) variable attribute on page 3-145.
3.33 __attribute__((section("name"))) variable attribute on page 3-146.
3.35 __attribute__((unused)) variable attribute on page 3-148.
3.34 __attribute__((used)) variable attribute on page 3-147.
3.36 __attribute__((weak)) variable attribute on page 3-149.
3.37 __attribute__((weakref("target"))) variable attribute on page 3-150.

3.29 __attribute__((alias)) variable attribute

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

Aliases must be declared in the same translation unit as the definition of the original variable.

Note

Aliases cannot be specified 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:

```
        .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
}
```

3.30 __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));
```

3.31 __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' has been explicitly marked deprecated here
  extern int deprecated_var __attribute__((deprecated));
  ^
1 warning generated.
```

3.32 __attribute__((packed)) variable attribute

You can specify the `packed` variable attribute on fields that are members of a structure or union. It specifies that a member field has the smallest possible alignment. That is, one byte for a variable field, and one bit for a bitfield, unless you specify a larger value with the `aligned` attribute.

Example

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

Note

You must access a `packed` member of a structure or union directly from a variable of the structure or union. Taking the address of such a member produces a normal pointer which might be unaligned. The compiler assumes that the pointer is aligned. Dereferencing such a pointer can be unsafe even when unaligned accesses are supported by the target, because certain instructions always require word-aligned addresses.

Note

If you take the address of a packed member, in most cases, the compiler generates a warning.

Related references

3.30 __attribute__((aligned)) variable attribute on page 3-143.

3.33 __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.

To place ZI data in a named section, the section must start with the prefix `.bss.`. Non-ZI data cannot be placed in a section named `.bss.`

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")));
```

Note

Section names must be unique. You must not use the same section name for different section types. If you use the same section name for different section types, then the compiler merges the sections into one and gives the section the type of whichever function or variable is first allocated to that section.

3.34 __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, by 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
```

3.35 __attribute__((unused)) variable attribute

The compiler can warn if a variable is declared but is never referenced. The __attribute__((unused)) attribute informs the compiler to expect an unused variable, 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 used to warn about most unused variables, but 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

3.19 __attribute__((unused)) function attribute on page 3-132.

3.36 __attribute__((weak)) variable attribute

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

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

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

3.37 __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 Intrinsics

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

To use these intrinsics, your source file must contain `#include <arm_compat.h>`.

It contains the following sections:

4.1 __breakpoint intrinsic

This intrinsic inserts a `BKPT` instruction into the instruction stream generated by the compiler.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for AArch32.

It enables you to include a breakpoint instruction in your C or C++ code.

Syntax

```
void __breakpoint(int val)
```

Where:

val
is a compile-time constant integer whose range is:

`0 ... 65535`
if you are compiling source as ARM code

`0 ... 255`
if you are compiling source as Thumb code.

Errors

The `__breakpoint` intrinsic is not available when compiling for a target that does not support the `BKPT` instruction. The compiler generates an error in this case.

Example

```
void func(void)
{
    ...
    __breakpoint(0xF02C);
    ...
}
```

4.2 __current_pc intrinsic

This intrinsic enables you to determine the current value of the program counter at the point in your program where the intrinsic is used.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for AArch32.

Syntax

```
unsigned int __current_pc(void)
```

Return value

The `__current_pc` intrinsic returns the current value of the program counter at the point in the program where the intrinsic is used.

4.3 __current_sp intrinsic

This intrinsic returns the value of the stack pointer at the current point in your program.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for AArch32.

Syntax

```
unsigned int __current_sp(void)
```

Return value

The `__current_sp` intrinsic returns the current value of the stack pointer at the point in the program where the intrinsic is used.

4.4 __disable_fiq intrinsic

This intrinsic disables FIQ interrupts.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for AArch32.

Note

Typically, this intrinsic disables FIQ interrupts by setting the F-bit in the CPSR. However, for v7-M and v8-M.mainline, it sets the fault mask register (FAULTMASK). This intrinsic is not supported for v6-M and v8-M.baseline.

Syntax

```
int __disable_fiq(void)
```

Usage

`int __disable_fiq(void);` disables fast interrupts and returns the value the FIQ interrupt mask has in the `PSR` before disabling interrupts.

Return value

`int __disable_fiq(void);` returns the value the FIQ interrupt mask has in the `PSR` before disabling FIQ interrupts.

Restrictions

The `__disable_fiq` intrinsic can only be executed in privileged modes, that is, in non-user modes. In User mode this intrinsic does not change the interrupt flags in the `CPSR`.

Example

```
void foo(void)
{
    int was_masked = __disable_fiq();
    /* ... */
    if (!was_masked)
        __enable_fiq();
}
```

4.5 __disable_irq intrinsic

This intrinsic disables IRQ interrupts.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for AArch32.

Note

Typically, this intrinsic disables IRQ interrupts by setting the I-bit in the CPSR. However, for M-profile it sets the exception mask register (PRIMASK).

Syntax

```
int __disable_irq(void)
```

Usage

`int __disable_irq(void);` disables interrupts and returns the value the IRQ interrupt mask has in the `PSR` before disabling interrupts.

Return value

`int __disable_irq(void);` returns the value the IRQ interrupt mask has in the `PSR` before disabling IRQ interrupts.

Example

```
void foo(void)
{
    int was_masked = __disable_irq();
    /* ... */
    if (!was_masked)
        __enable_irq();
}
```

Restrictions

The `__disable_irq` intrinsic can only be executed in privileged modes, that is, in non-user modes. In User mode this intrinsic does not change the interrupt flags in the `CPSR`.

4.6 __enable_fiq intrinsic

This intrinsic enables FIQ interrupts.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for AArch32.

Note

Typically, this intrinsic enables FIQ interrupts by clearing the F-bit in the CPSR. However, for v7-M and v8-M.mainline, it clears the fault mask register (FAULTMASK). This intrinsic is not supported in v6-M and v8-M.baseline.

Syntax

```
void __enable_fiq(void)
```

Restrictions

The `__enable_fiq` intrinsic can only be executed in privileged modes, that is, in non-user modes. In User mode this intrinsic does not change the interrupt flags in the `CPSR`.

4.7 __enable_irq intrinsic

This intrinsic enables IRQ interrupts.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for AArch32.

Note

Typically, this intrinsic enables IRQ interrupts by clearing the I-bit in the CPSR. However, for Cortex M-profile processors, it clears the exception mask register (PRIMASK).

Syntax

```
void __enable_irq(void)
```

Restrictions

The `__enable_irq` intrinsic can only be executed in privileged modes, that is, in non-user modes. In User mode this intrinsic does not change the interrupt flags in the `CPSR`.

4.8 __force_stores intrinsic

This intrinsic causes all variables that are visible outside the current function, such as variables that have pointers to them passed into or out of the function, to be written back to memory if they have been changed.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for AArch32.

This intrinsic also acts as a `__schedule_barrier` intrinsic.

Syntax

```
void __force_stores(void)
```

4.9 __memory_changed intrinsic

This intrinsic causes the compiler to behave as if all C objects had their values both read and written at that point in time.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for AArch32.

The compiler ensures that the stored value of each C object is correct at that point in time and treats the stored value as unknown afterwards.

This intrinsic also acts as a `__schedule_barrier` intrinsic.

Syntax

```
void __memory_changed(void)
```

4.10 __schedule_barrier intrinsic

This intrinsic creates a special sequence point that prevents operations with side effects from moving past it under all circumstances. Normal sequence points allow operations with side effects past if they do not affect program behavior. Operations without side effects are not restricted by the intrinsic, and the compiler can move them past the sequence point.

Operations with side effects cannot be reordered above or below the `__schedule_barrier` intrinsic. To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for AArch32.

Unlike the `__force_stores` intrinsic, the `__schedule_barrier` intrinsic does not cause memory to be updated. The `__schedule_barrier` intrinsic is similar to the `__nop` intrinsic, only differing in that it does not generate a `NOP` instruction.

Syntax

```
void __schedule_barrier(void)
```

4.11 __semihost intrinsic

This intrinsic inserts an SVC or BKPT instruction into the instruction stream generated by the compiler. It enables you to make semihosting calls from C or C++ that are independent of the target architecture.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for AArch32.

Syntax

```
int __semihost(int val, const void *ptr)
```

Where:

val
: Is the request code for the semihosting request.

ptr
: Is a pointer to an argument/result block.

Return value

The results of semihosting calls are passed either as an explicit return value or as a pointer to a data block.

Usage

Use this intrinsic from C or C++ to generate the appropriate semihosting call for your target and instruction set:

SVC `0x123456`
: In ARM state, excluding M-profile architectures.

SVC `0xAB`
: In Thumb state, excluding M-profile architectures. This behavior is not guaranteed on *all* debug targets from ARM or from third parties.

HLT `0xF000`
: In ARM state, excluding M-profile architectures.

HLT `0x3C`
: In Thumb state, excluding M-profile architectures.

BKPT `0xAB`
: For M-profile architectures (Thumb only).

Implementation

For ARM processors that are not Cortex-M profile, semihosting is implemented using the SVC or HLT instruction. For Cortex M-profile processors, semihosting is implemented using the BKPT instruction.

To use HLT-based semihosting, you must define the pre-processor macro `__USE_HLT_SEMIHOSTING` before `#include <arm_compat.h>`. By default, ARM Compiler emits SVC instructions rather than HLT instructions for semihosting calls. If you define this macro, `__USE_HLT_SEMIHOSTING`, then ARM Compiler emits HLT instructions rather than SVC instructions for semihosting calls.

The presence of this macro, `__USE_HLT_SEMIHOSTING`, does not affect the M-profile architectures which still use BKPT for semihosting.

Example

```
char buffer[100];
...
void foo(void)
{
    __semihost(0x01, (const void *)buffer);
}
```

Compiling this code with the option `-mthumb` shows the generated SVC instruction:

```
foo:
    ...
    MOVW     r0, :lower16:buffer
    MOVT     r0, :upper16:buffer
    ...
    SVC      #0xab
    ...

buffer:
    .zero    100
    .size    buffer, 100
```

Related information

Using the C and C++ libraries with an application in a semihosting environment.

4.12 __vfp_status intrinsic

This intrinsic reads or modifies the FPSCR.

To use this intrinsic, your source file must contain `#include <arm_compat.h>`. This is only available for AArch32.

Syntax

```
unsigned int __vfp_status(unsigned int mask, unsigned int flags)
```

Usage

Use this intrinsic to read or modify the flags in FPSCR.

The intrinsic returns the value of FPSCR, unmodified, if *mask* and *flags* are 0.

You can clear, set, or toggle individual flags in FPSCR using the bits in *mask* and *flags*, as shown in the following table. The intrinsic returns the modified value of FPSCR if *mask* and *flags* are not both 0.

Table 4-1 Modifying the FPSCR flags

mask bit	*flags* bit	Effect on FPSCR flag
0	0	Does not modify the flag
0	1	Toggles the flag
1	1	Sets the flag
1	0	Clears the flag

Note

If you want to read or modify only the exception flags in FPSCR, then ARM recommends that you use the standard C99 features in `<fenv.h>`.

Errors

The compiler generates an error if you attempt to use this intrinsic when compiling for a target that does not have VFP.

Chapter 5
Compiler-specific Pragmas

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

It contains the following sections:

5.1 #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.

5.2 #pragma clang diagnostic

Allows you to suppress, enable, or change the severity of specific diagnostic messages from within your code.

For example, you can suppress a particular diagnostic message when compiling one specific function.

Note

Alternatively, you can use the command-line option, `-Wname`, to suppress or change the severity of messages, but the change applies for the entire compilation.

#pragma clang diagnostic ignored

```
#pragma clang diagnostic ignored "-Wname"
```

This pragma disables the diagnostic message specified by *name*.

#pragma clang diagnostic warning

```
#pragma clang diagnostic warning "-Wname"
```

This pragma sets the diagnostic message specified by *name* to warning severity.

#pragma clang diagnostic error

```
#pragma clang diagnostic error "-Wname"
```

This pragma sets the diagnostic message specified by *name* to error severity.

#pragma clang diagnostic fatal

```
#pragma clang diagnostic fatal "-Wname"
```

This pragma sets the diagnostic message specified by *name* to fatal error severity. Fatal error causes compilation to fail without processing the rest of the file.

#pragma clang diagnostic push, #pragma clang diagnostic pop

```
#pragma clang diagnostic push
#pragma clang diagnostic pop
```

`#pragma clang diagnostic push` saves the current pragma diagnostic state so that it can restored later.

`#pragma clang diagnostic pop` restores the diagnostic state that was previously saved using `#pragma clang diagnostic push`.

Examples of using pragmas to control diagnostics

The following example shows four identical functions, `foo1()`, `foo2()`, `foo3()`, and `foo4()`. All these functions would normally provoke diagnostic message `warning: multi-character character constant [-Wmultichar]` on the source lines `char c = (char) 'ab';`

Using pragmas, you can suppress or change the severity of these diagnostic messages for individual functions.

For `foo1()`, the current pragma diagnostic state is pushed to the stack and `#pragma clang diagnostic ignored` suppresses the message. The diagnostic message is then re-enabled by `#pragma clang diagnostic pop`.

For `foo2()`, the diagnostic message is not suppressed because the original pragma diagnostic state has been restored.

For `foo3()`, the message is initially suppressed by the preceding `#pragma clang diagnostic ignored "-Wmultichar"`, however, the message is then re-enabled as an error, using `#pragma clang diagnostic error "-Wmultichar"`. The compiler therefore reports an error in `foo3()`.

For `foo4()`, the pragma diagnostic state is restored to the state saved by the preceding `#pragma clang diagnostic push`. This state therefore includes `#pragma clang diagnostic ignored "-Wmultichar"` and therefore the compiler does not report a warning in `foo4()`.

```
#pragma clang diagnostic push
#pragma clang diagnostic ignored "-Wmultichar"
void foo1( void )
{
    /* Here we do not expect a diagnostic message, because it is suppressed by #pragma clang
diagnostic ignored "-Wmultichar". */
    char c = (char) 'ab';
}
#pragma clang diagnostic pop

void foo2( void )
{
    /* Here we expect a warning, because the suppression was inside push and then the
diagnostic message was restored by pop. */
    char c = (char) 'ab';
}

#pragma clang diagnostic ignored "-Wmultichar"
#pragma clang diagnostic push
void foo3( void )
{
    #pragma clang diagnostic error "-Wmultichar"
    /* Here, the diagnostic message is elevated to error severity. */
    char c = (char) 'ab';
}
#pragma clang diagnostic pop

void foo4( void )
{
    /* Here, there is no diagnostic message because the restored diagnostic state only
includes the #pragma clang diagnostic ignored "-Wmultichar".
       It does not include the #pragma clang diagnostic error "-Wmultichar" that is within
the push and pop pragmas. */
    char c = (char) 'ab';
}
```

Diagnostic messages use the pragma state that is present at the time they are generated. If you use pragmas to control a diagnostic message in your code, you must be aware of when, in the compilation process, that diagnostic message is generated.

If a diagnostic message for a function, `functionA`, is only generated after all the functions have been processed, then the compiler controls this diagnostic message using the pragma diagnostic state that is present after processing all the functions. This diagnostic state might be different from the diagnostic state immediately before or within the definition of `functionA`.

Related references

1.68 -W on page 1-94.

5.3 #pragma clang section

Specifies one or more section names to be used for subsequent functions, global variables, or static variables within a compilation unit.

Syntax

`#pragma clang section [section_type_list]`

Where:

section_type_list
specifies an optional list of section names to be used for subsequent functions, global variables, or static variables. The syntax of *section_type_list* is:

`section_type="name"[ section_type="name"]`

You can revert to the default section name by specifying an empty string, `""`, for *name*.

Valid section types are:

- `bss`.
- `data`.
- `rodata`.
- `text`.

Usage

Use `#pragma arm section [section_type_list]` to place functions and variables in separate named sections. You can then use the scatter-loading description file to locate these at a particular address in memory.

- If you specify a section name with `__attribute__((section("myname")))`, then the attribute name has priority over any applicable section name that you specify with `#pragma clang section`.
- `#pragma clang section` has priority over the `-ffunction-section` and `-fdata-section` command-line options.
- Global variables, including basic types, arrays, and struct that are initialized to zero are placed in the `.bss` section. For example, `int x = 0;`.
- `armclang` does not try to infer the type of section from the name. For example, assigning a section `.bss.mysec` does not mean it is placed in a `.bss` section.
- If you specify the `-ffunction-section` and `-fdata-section` command-line options, then each global variable is in a unique section.

Example

```
int x1 = 5;                     // Goes in .data section (default)
int y1;                         // Goes in .bss section (default)
const int z1 = 42;              // Goes in .rodata section (default)
char *s1 = "abc1";              // s1 goes in .data section (default). String "abc1" goes
in .conststring section.

#pragma clang section bss="myBSS" data="myData" rodata="myRodata"
int x2 = 5;                     // Goes in myData section.
int y2;                         // Goes in myBss section.
const int z2 = 42;              // Goes in myRodata section.
char *s2 = "abc2";              // s2 goes in myData section. String "abc2" goes
in .conststring section.

#pragma arm section rodata=""   // Use default name for rodata section.
int x3 = 5;                     // Goes in myData section.
int y3;                         // Goes in myBss section.
const int z3 = 42;              // Goes in .rodata section (default).
char *s3 = "abc3";              // s3 goes in myData section. String "abc3" goes
in .conststring section.

#pragma clang section text="myText"
int add1(int x)                 // Goes in myText section.
{
    return x+1;
}
```

```
#pragma clang section bss="" data="" text="" // Use default name for bss, data, and text
sections.
```

5.4 #pragma once

Enable 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, ARM recommends using `#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.

5.5 #pragma pack(...)

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. You can optionally push and restore alignment settings to an internal stack.

Note

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

Syntax

```
#pragma pack(n)

#pragma pack(push[,n])

#pragma pack(pop)
```

Where:

n
: Is the alignment in bytes, valid alignment values being `1`, `2`, `4` and `8`. If omitted, sets the alignment to the one that was in effect when compilation started.

`push[,n]`
: Pushes the current alignment setting on an internal stack and then optionally sets the new alignment.

`pop`
: Restores the alignment setting to the one saved at the top of the internal stack, then removes that stack entry.

  Note

  `#pragma pack([n])` does not influence this internal stack. Therefore, it is possible to have `#pragma pack(push)` followed by multiple `#pragma pack(n)` instances, then finalized by a single `#pragma pack(pop)`.

Default

The default is the alignment that was in effect when compilation started.

Example

This example shows 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:

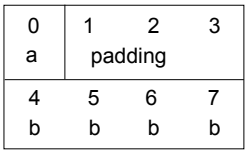

Figure 5-1 Nonpacked structure S

The layout of `SP` is:

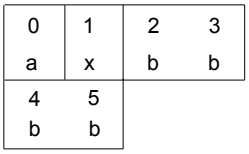

Figure 5-2 Packed structure SP

Note

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

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

5.6 #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` will not unroll a loop if the number of iterations is not known at compile time.

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. This pragma can be used 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.

5.7 #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 6
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:

6.1 ACLE support

ARM Compiler 6 supports the ARM C Language Extensions 2.0 with a few exceptions.

ARM Compiler 6 does not support:

- `__attribute__((target("arm")))` attribute.
- `__attribute__((target("thumb")))` attribute.
- `__ARM_ALIGN_MAX_PWR` macro.
- `__ARM_ALIGN_MAX_STACK_PWR` macro.
- `__cls` intrinsic.
- `__clsl` intrinsic.
- `__clsll` intrinsic.
- `__saturation_occurred` intrinsic.
- `__set_saturation_occurred` intrinsic.
- `__ignore_saturation` intrinsic.
- Patchable constants.
- 16-bit multiplication intrinsics.
- Floating-point data-processing intrinsics.
- Intrinsics for the 32-bit SIMD instructions introduced in ARMv6.

ARM Compiler 6 does not model the state of the Q (saturation) flag correctly in all situations.

Related information

ARM C Language Extensions.

6.2 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 `-E -dM` to see the values of predefined macros.

Macros beginning with `__ARM_` are defined by the *ARM C Language Extensions 2.0* (ACLE 2.0).

Note

`armclang` does not fully implement ACLE 2.0.

Table 6-1 Predefined macros

Name	Value	When defined
`__APCS_ROPI`	`1`	Set when you specify the `-fropi` option.
`__APCS_RWPI`	`1`	Set when you specify the `-frwpi` option.
`__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_BIG_ENDIAN`	`-`	Set if compiling for a big-endian target.
`__ARM_FEATURE_CLZ`	`1`	Set to 1 if the `CLZ` (count leading zeroes) instruction is supported in hardware.

Table 6-1 Predefined macros (continued)

Name	Value	When defined
__ARM_FEATURE_CMSE	*num*	Indicates the availability of the ARMv8-M Security Extensions related extensions: 0 The ARMv8-M TT instruction is not available. 1 The TT instruction is available. It is not part of ARMv8-M Security Extensions, but is closely related. 3 The ARMv8-M Security Extensions for secure executable files is available. This implies that the TT instruction is available. See *6.5 TT instruction intrinsics* on page 6-185 for more information.
__ARM_FEATURE_CRC32	1	Set to 1 if the target has CRC extension.
__ARM_FEATURE_CRYPTO	1	Set to 1 if the target has cryptographic extension.
__ARM_FEATURE_DIRECTED_ROUNDING	1	Set to 1 if the directed rounding and conversion vector instructions are supported.Only available when __ARM_ARCH >= 8.
__ARM_FEATURE_DSP	1	Set for 32-bit targets only. Set to 1 if DSP instructions are supported. This feature also implies support for the Q flag. **Note** This macro is deprecated in ACLE 2.0 for A-profile. It is fully supported for M and R-profiles.
__ARM_FEATURE_IDIV	1	Set to 1 if the target supports 32-bit signed and unsigned integer division in all available instruction sets.
__ARM_FEATURE_FMA	1	Set to 1 if the target supports fused floating-point multiply-accumulate.
__ARM_FEATURE_NUMERIC_MAXMIN	1	Set to 1 if the target supports floating-point maximum and minimum instructions. Only available when __ARM_ARCH >= 8.
__ARM_FEATURE_QBIT	1	Set for 32-bit targets only. Set to 1 if the Q (saturation) flag exists. **Note** This macro is deprecated in ACLE 2.0 for A-profile.
__ARM_FEATURE_SAT	1	Set for 32-bit targets only. Set to 1 if the SSAT and USAT instructions are supported. This feature also implies support for the Q flag. **Note** This macro is deprecated in ACLE 2.0 for A-profile.

Table 6-1 Predefined macros (continued)

Name	Value	When defined
__ARM_FEATURE_SIMD32	1	Set for 32-bit targets only. Set to 1 if the target supports 32-bit SIMD instructions. ——— **Note** ——— This macro is deprecated in ACLE 2.0 for A-profile, use NEON intrinsics instead. ———
__ARM_FEATURE_UNALIGNED	1	Set to 1 if the target supports unaligned access in hardware.
__ARM_FP	*val*	Set if hardware floating-point is available. Bits 1-3 indicate the supported floating-point precision levels. The other bits are reserved. • Bit 1 - half precision (16-bit). • Bit 2 - single precision (32-bit). • Bit 3 - double precision (64-bit). These bits can be bitwise or-ed together. Permitted values include: • `0x04` for single-support. • `0x0C` for single- and double-support. • `0x0E` for half-, single-, and double-support.
__ARM_FP_FAST	1	Set if `-ffast-math` or `-ffp-mode=fast` is specified.
__ARM_NEON	1	Set to 1 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	*val*	This is the same as `__ARM_FP`, except that the bit to indicate double-precision is not set for AArch32. Double-precision is always set for AArch64.
__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 the default procedure calling standard for the translation unit conforms to the VFP PCS. That is, `-mfloat-abi=hard`.
__ARM_SIZEOF_MINIMAL_ENUM	*value*	Specifies the size of the minimal enumeration type. Set to either 1 or 4 depending on whether `-fshort-enums` is specified or not.
__ARM_SIZEOF_WCHAR_T	value	Specifies the size of `wchar` in bytes. Set to 2 if `-fshort-wchar` is specified, or 4 if `-fno-short-wchar` is specified. ——— **Note** ——— The default size is 4, because `-fno-short-wchar` is set by default. ———

Table 6-1 Predefined macros (continued)

Name	Value	When defined
__ARMCOMPILER_VERSION	*Mmmuuxx*	Always set. Specifies the version number of the compiler, `armclang`. The format is *Mmmuuxx*, where: • *M* is the major version number, `6`. • *mm* is the minor version number. • *uu* is the update number. • *xx* is reserved for ARM internal use. You can ignore this for the purposes of checking whether the current release is a specific version or within a range of versions. For example, version 6.3 update 1 is displayed as `6030154`, where `54` is a number for ARM internal use.
__ARMCC_VERSION	*Mmmuuxx*	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.
__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 to 1 when compiling with software floating-point on 32-bit targets. Set to 0 otherwise.

Table 6-1 Predefined macros (continued)

Name	Value	When defined
`__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

1.66 --version_number on page 1-92.
1.60 -std on page 1-86.
1.55 -O on page 1-80.
1.61 --target on page 1-87.
1.41 -marm on page 1-62.
1.51 -mthumb on page 1-76.

6.3 Inline functions

Inline functions offer a trade-off between code size and performance. By default, the compiler decides whether to inline functions.

With regards to optimization, by default the compiler optimizes for performance with respect to time. If the compiler decides to inline a function, it makes sure to avoid large code growth. When compiling to restrict code size, through the use of `-Oz` or `-Os`, the compiler makes sensible decisions about inlining and aims to keep code size to a minimum.

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 short functions.

Note

The default semantic rules for C-source code follow C99 rules. For inlining, it means that when you suggest 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, because 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 2-109.
1.60 -std on page 1-86.
3.2 __attribute__((always_inline)) function attribute on page 3-115.

6.4 Half-precision floating-point number format

ARM Compiler supports the half-precision floating-point `__fp16` type.

Half-precision is a floating-point format that occupies 16 bits. Architectures that support half-precision floating-point numbers include:

- The ARMv8 architecture.
- The ARMv7 FPv5 architecture.
- The ARMv7 VFPv4 architecture.
- The ARMv7 VFPv3 architecture (as an optional extension).

If the target hardware does not support half-precision floating-point numbers, the compiler uses the floating-point library `fplib` to provide software support for half-precision.

Note

The `__fp16` type is a storage format only. For purposes of arithmetic and other operations, `__fp16` values in C or C++ expressions are automatically promoted to `float`.

Half-precision floating-point format

ARM Compiler uses the half-precision binary floating-point format defined by IEEE 754r, a revision to the IEEE 754 standard:

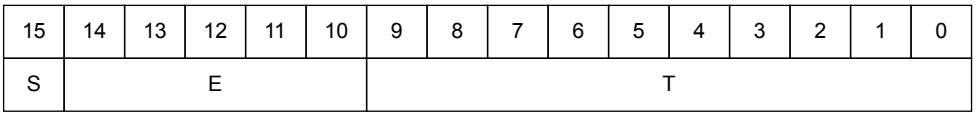

Figure 6-1 IEEE half-precision floating-point format

Where:

```
S (bit[15]):      Sign bit
E (bits[14:10]):  Biased exponent
T (bits[9:0]):    Mantissa.
```

The meanings of these fields are as follows:

```
IF E==31:
   IF T==0: Value = Signed infinity
   IF T!=0: Value = Nan
             T[9] determines Quiet or Signalling:
                  0: Quiet NaN
                  1: Signalling NaN
IF 0<E<31:
   Value = (-1)^S x 2^(E-15) x (1 + (2^(-10) x T))
IF E==0:
   IF T==0: Value = Signed zero
   IF T!=0: Value = (-1)^S x 2^(-14) x (0 + (2^(-10) x T))
```

Note

See the *ARM C Language Extensions* for more information.

Related information

ARM C Language Extensions.

6.5 TT instruction intrinsics

Intrinsics are available to support TT instructions depending on the value of the predefined macro __ARM_FEATURE_CMSE.

TT intrinsics

The following table describes the TT intrinsics that are available when __ARM_FEATURE_CMSE is set to either 1 or 3:

Intrinsic	Description
`cmse_address_info_t cmse_TT(void *p)`	Generates a TT instruction.
`cmse_address_info_t cmse_TT_fptr(p)`	Generates a TT instruction. The argument p can be any function pointer type.
`cmse_address_info_t cmse_TTT(void *p)`	Generates a TT instruction with the T flag.
`cmse_address_info_t cmse_TTT_fptr(p)`	Generates a TT instruction with the T flag. The argument p can be any function pointer type.

When __ARM_BIG_ENDIAN is not set, the result of the intrinsics is returned in the following C type:

```
typedef union {
    struct cmse_address_info {
        unsigned mpu_region:8;
        unsigned :8;
        unsigned mpu_region_valid:1;
        unsigned :1;
        unsigned read_ok:1;
        unsigned readwrite_ok:1;
        unsigned :12;
    } flags;
    unsigned value;
} cmse_address_info_t;
```

When __ARM_BIG_ENDIAN is set, the bit-fields in the type are reversed such that they have the same bit-offset as little-endian systems following the rules specified by *Procedure Call Standard for the ARM® Architecture*.

TT intrinsics for ARMv8-M Security Extensions

The following table describes the TT intrinsics for ARMv8-M Security Extensions that are available when __ARM_FEATURE_CMSE is set to 3:

Intrinsic	Description
`cmse_address_info_t cmse_TTA(void *p)`	Generates a TT instruction with the A flag.
`cmse_address_info_t cmse_TTA_fptr(p)`	Generates a TT instruction with the A flag. The argument p can be any function pointer type.
`cmse_address_info_t cmse_TTAT(void *p)`	Generates a TT instruction with the T and A flag.
`cmse_address_info_t cmse_TTAT_fptr(p)`	Generates a TT instruction with the T and A flag. The argument p can be any function pointer type.

When __ARM_BIG_ENDIAN is not set, the result of the intrinsics is returned in the following C type:

```
typedef union {
    struct cmse_address_info {
        unsigned mpu_region:8;
```

```
        unsigned sau_region:8;
        unsigned mpu_region_valid:1;
        unsigned sau_region_valid:1;
        unsigned read_ok:1;
        unsigned readwrite_ok:1;
        unsigned nonsecure_read_ok:1;
        unsigned nonsecure_readwrite_ok:1;
        unsigned secure:1;
        unsigned idau_region_valid:1;
        unsigned idau_region:8;
    } flags;
    unsigned value;
} cmse_address_info_t;
```

When `__ARM_BIG_ENDIAN` is set, the bit-fields in the type are reversed such that they have the same bit-offset as little-endian systems following the rules specified by *Procedure Call Standard for the ARM® Architecture*.

In the Secure state, the `TT` instruction returns the *Security Attribute Unit* (SAU) and *Implementation Defined Attribute Unit* (IDAU) configuration and recognizes the `A` flag.

Address range check intrinsic

Checking the result of the `TT` instruction on an address range is essential for programming in C. It is needed to check permissions on objects larger than a byte. You can use the address range check intrinsic to perform permission checks on C objects.

The syntax of this intrinsic is:

```
void *cmse_check_address_range(void *p, size_t size, int flags)
```

The intrinsic checks the address range from `p` to `p + size – 1`.

The address range check fails if `p + size - 1 < p`.

Some SAU, IDAU and MPU configurations block the efficient implementation of an address range check. This intrinsic operates under the assumption that the configuration of the SAU, IDAU, and MPU is constrained as follows:

- An object is allocated in a single region.
- A stack is allocated in a single region.

These points imply that a region does not overlap other regions.

The `TT` instruction returns an SAU, IDAU and MPU region number. When the region numbers of the start and end of the address range match, the complete range is contained in one SAU, IDAU, and MPU region. In this case two `TT` instructions are executed to check the address range.

Regions are aligned at 32-byte boundaries. If the address range fits in one 32-byte address line, a single `TT` instruction suffices. This is the case when the following constraint holds:

`(`*`p`* `mod 32) +` *`size`* `<= 32`

The address range check intrinsic fails if the range crosses any MPU region boundary.

The `flags` parameter of the address range check consists of a set of values defined by the macros shown in the following table:

Macro	Value	Description
(No macro)	`0`	The `TT` instruction without any flag is used to retrieve the permissions of an address, returned in a `cmse_address_info_t` structure.
`CMSE_MPU_UNPRIV`	`4`	Sets the `T` flag on the `TT` instruction used to retrieve the permissions of an address. Retrieves the unprivileged mode access permissions.

(continued)

Macro	Value	Description
CMSE_MPU_READWRITE	1	Checks if the permissions have the readwrite_ok field set.
CMSE_MPU_READ	8	Checks if the permissions have the read_ok field set.

The address range check intrinsic returns p on a successful check, and NULL on a failed check. The check fails if any other value is returned that is not one of those listed in the table, or is not a combination of those listed.

ARM recommends that you use the returned pointer to access the checked memory range. This generates a data dependency between the checked memory and all its subsequent accesses and prevents these accesses from being scheduled before the check.

The following intrinsic is defined when the __ARM_FEATURE_CMSE macro is set to 1:

Intrinsic	Description
cmse_check_pointed_object(p, f)	Returns the same value as cmse_check_address_range(p, sizeof(*p), f)

ARM recommends that the return type of this intrinsic is identical to the type of parameter p.

Address range check intrinsic for ARMv8-M Security Extensions

The semantics of the intrinsic cmse_check_address_range() are extended to handle the extra flag and fields introduced by the ARMv8-M Security Extensions.

The address range check fails if the range crosses any SAU or IDAU region boundary.

If the macro __ARM_FEATURE_CMSE is set to 3, the values accepted by the flags parameter are extended with the values defined in the following table:

Macro	Value	Description
CMSE_AU_NONSECURE	2	Checks if the permissions have the secure field unset.
CMSE_MPU_NONSECURE	16	Sets the A flag on the TT instruction used to retrieve the permissions of an address.
CMSE_NONSECURE	18	Combination of CMSE_AU_NONSECURE and CMSE_MPU_NONSECURE.

Related references

6.2 Predefined macros on page 6-178.

6.6 Non-secure function pointer intrinsics

A non-secure function pointer is a function pointer that has its LSB unset.

The following table describes the non-secure function pointer intrinsics that are available when `__ARM_FEATURE_CMSE` is set to `3`:

Intrinsic	Description
`cmse_nsfptr_create(p)`	Returns the value of `p` with its LSB cleared. The argument `p` can be any function pointer type. ARM recommends that the return type of this intrinsic is identical to the type of its argument.
`cmse_is_nsfptr(p)`	Returns non-zero if `p` has LSB unset, zero otherwise. The argument `p` can be any function pointer type.

Example

The following example shows how to use these intrinsics:

```
#include <arm_cmse.h>
typedef void __attribute__((cmse_nonsecure_call)) nsfunc(void);
void default_callback(void) { … }

// fp can point to a secure function or a non-secure function
nsfunc *fp = (nsfunc *) default_callback;           // secure function pointer

void __attribute__((cmse_nonsecure_entry)) entry(nsfunc *callback) {
    fp = cmse_nsfptr_create(callback);  // non-secure function pointer
}

void call_callback(void) {
    if (cmse_is_nsfptr(fp)) fp();       // non-secure function call
    else ((void (*)(void)) fp)();       // normal function call
}
```

Related references

3.3 __attribute__((cmse_nonsecure_call)) function attribute on page 3-116.

3.4 __attribute__((cmse_nonsecure_entry)) function attribute on page 3-117.

Related information

Building Secure and Non-secure Images Using ARMv8-M Security Extensions.

Chapter 7
Standard C Implementation Definition

Provides information required by the ISO C standard for conforming C implementations.

It contains the following sections:

7.1 Implementation Definition

Appendix J of the ISO C standard (ISO/IEC 9899:2011 (E)) collates information about portability issues. Sub-clause J3 lists the behavior that each implementation must document. The following topics correspond to the relevant sections of sub-clause J3. They describe aspects of the ARM C Compiler and C library, not defined by the ISO C standard, that are implementation-defined. Whenever the implementation-defined behavior of the ARM C compiler or the C library can be altered and tailored to the execution environment by reimplementing certain functions, that behavior is described as "depends on the environment".

Related references

7.2 Translation on page 7-191.
7.3 Translation limits on page 7-192.
7.4 Environment on page 7-194.
7.5 Identifiers on page 7-196.
7.6 Characters on page 7-197.
7.7 Integers on page 7-199.
7.8 Floating-point on page 7-200.
7.9 Arrays and pointers on page 7-201.
7.10 Hints on page 7-202.
7.11 Structures, unions, enumerations, and bitfields on page 7-203.
7.12 Qualifiers on page 7-204.
7.13 Preprocessing directives on page 7-205.
7.14 Library functions on page 7-206.
7.15 Architecture on page 7-211.

7.2 Translation

Describes implementation-defined aspects of the ARM C compiler and C library relating to translation, as required by the ISO C standard.

How a diagnostic is identified (3.10, 5.1.1.3).

Diagnostic messages that the compiler produces are of the form:

```
source-file:line-number:char-number: description [diagnostic-flag]
```

Here:

description

Is a text description of the error.

diagnostic-flag

Is an optional diagnostic flag of the form `-Wflag`, only for messages that can be suppressed.

Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

Each nonempty sequence of white-space characters, other than new-line, is replaced by one space character.

7.3 Translation limits

Describes implementation-defined aspects of the ARM C compiler and C library relating to translation, as required by the ISO C standard.

Section *5.2.4.1 Translation limits* of the ISO/IEC 9899:2011 standard requires minimum translation limits that a conforming compiler must accept. The following table gives a summary of these limits. In this table, a limit of *memory* indicates that ARM Compiler 6 imposes no limit, other than that imposed by the available memory.

Table 7-1 Translation limits

Description	Translation limit
Nesting levels of block.	256 (can be increased using the `-fbracket-depth` option.)
Nesting levels of conditional inclusion.	*memory*
Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or void type in a declaration.	*memory*
Nesting levels of parenthesized declarators within a full declarator.	256 (can be increased using the `-fbracket-depth` option.)
Nesting levels of parenthesized expressions within a full expression.	256 (can be increased using the `-fbracket-depth` option.)
Significant initial characters in an internal identifier or a macro name.	*memory*
Significant initial characters in an external identifier.	*memory*
External identifiers in one translation unit.	*memory*
Identifiers with block scope declared in one block.	*memory*
Macro identifiers simultaneously defined in one preprocessing translation unit.	*memory*
Parameters in one function definition.	*memory*
Arguments in one function call.	*memory*
Parameters in one macro definition.	*memory*
Arguments in one macro invocation.	*memory*
Characters in a logical source line.	*memory*
Characters in a string literal.	*memory*
Bytes in an object.	*SIZE_MAX*
Nesting levels for `#include` files.	*memory*
Case labels for a switch statement.	*memory*
Members in a single structure or union.	*memory*
Enumeration constants in a single enumeration.	*memory*
Levels of nested structure or union definitions in a single struct-declaration-list.	256 (can be increased using the `-fbracket-depth` option.)

Related references

1.7 -fbracket-depth=N on page 1-25.
7.15 Architecture on page 7-211.

7.4 Environment

Describes implementation-defined aspects of the ARM C compiler and C library relating to environment, as required by the ISO C standard.

The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2).

The compiler interprets the physical source file multibyte characters as UTF-8.

The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

When linking with microlib, the function `main()` must be declared to take no arguments and must not return.

The effect of program termination in a freestanding environment (5.1.2.1).

The function `exit()` is not supported by microlib and the function `main()` must not return.

An alternative manner in which the main function can be defined (5.1.2.2.1).

The main function can be defined in one of the following forms:

```
int main(void)
int main()
int main(int)
int main(int, char **)
int main(int,char **, char **)
```

The values given to the strings pointed to by the argv argument to main (5.1.2.2.1).

In the generic ARM library the arguments given to `main()` are the words of the command line not including input/output redirections, delimited by whitespace, except where the whitespace is contained in double quotes.

What constitutes an interactive device (5.1.2.3).

What constitutes an interactive device depends on the environment and the `_sys_istty` function. The standard I/O streams `stdin`, `stdout`, and `stderr` are assumed to be interactive devices. They are line-buffered at program startup, regardless of what `_sys_istty` reports for them. An exception is if they have been redirected on the command line.

Whether a program can have more than one thread of execution in a freestanding environment (5.1.2.4).

Depends on the environment. The microlib C library is not thread-safe.

The set of signals, their semantics, and their default handling (7.14).

The `<signal.h>` header defines the following signals:

Signal	Value	Semantics
`SIGABRT`	1	Abnormal termination
`SIGFPE`	2	Arithmetic exception
`SIGILL`	3	Illegal instruction execution
`SIGINT`	4	Interactive attention signal
`SIGSEGV`	5	Bad memory access
`SIGTERM`	6	Termination request
`SIGSTAK`	7	Stack overflow (obsolete)
`SIGRTRED`	8	Run-time redirection error
`SIGRTMEM`	9	Run-time memory error
`SIGUSR1`	10	Available for the user
`SIGUSR2`	11	Available for the user
`SIGPVFN`	12	Pure virtual function called
`SIGCPPL`	13	Not normally used
`SIGOUTOFHEAP`	14	`::operator new` or `::operator new[]` cannot allocate memory

The default handling of all recognized signals is to print a diagnostic message and call `exit()`.

Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).

No signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` correspond to a computational exception.

Signals for which the equivalent of `signal(sig, SIG_IGN);` is executed at program startup (7.14.1.1).

No signals are ignored at program startup.

The set of environment names and the method for altering the environment list used by the `getenv` function (7.22.4.6).

The default implementation returns `NULL`, indicating that no environment information is available.

The manner of execution of the string by the system function (7.22.4.8).

Depends on the environment. The default implementation of the function uses semihosting.

7.5 Identifiers

Describes implementation-defined aspects of the ARM C compiler and C library relating to identifiers, as required by the ISO C standard.

Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

Multibyte characters, whose UTF-8 decoded value falls within one of the ranges in Appendix D of ISO/IEC 9899:2011 are allowed in identifiers and correspond to the universal character name with the short identifier (as specified by ISO/IEC 10646) having the same numeric value.

The dollar character `$` is allowed in identifiers.

The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

There is no limit on the number of significant initial characters in an identifier.

7.6 Characters

Describes implementation-defined aspects of the ARM C compiler and C library relating to characters, as required by the ISO C standard.

The number of bits in a byte (3.6).

The number of bits in a byte is 8.

The values of the members of the execution character set (5.2.1).

The values of the members of the execution character set are all the code points defined by ISO/IEC 19646.

The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

Character escape sequences have the following values in the execution character set:

Escape sequence	Char value	Description
`\a`	`7`	Attention (bell)
`\b`	`8`	Backspace
`\t`	`9`	Horizontal tab
`\n`	`10`	New line (line feed)
`\v`	`11`	Vertical tab
`\f`	`12`	Form feed
`\r`	`13`	Carriage return

The value of a char object into which has been stored any character other than a member of the basic execution character set (6.2.5).

The value of a **`char`** object into which has been stored any character other than a member of the basic execution character set is the least significant 8 bits of that character, interpreted as unsigned.

Which of signed char or unsigned char has the same range, representation, and behavior as plain `char` (6.2.5, 6.3.1.1).

Data items of type **`char`** are unsigned by default. The type **`unsigned char`** has the same range, representation, and behavior as **`char`**.

The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

The execution character set is identical to the source character set.

The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

In C all character constants have type **`int`**. Up to four characters of the constant are represented in the integer value. The last character in the constant occupies the lowest-order byte of the integer value. Up to three preceding characters are placed at higher-order bytes. Unused bytes are filled with the `NUL` (`\0`) character.

The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

If a wide character constant contains more than one multibyte character, all but the last such character are ignored.

The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

Mapping of wide character constants to the corresponding wide character code is locale independent.

Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence (6.4.5).

Differently prefixed wide string literal tokens cannot be concatenated.

The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).

Mapping of the wide characters in a wide string literal into the corresponding wide character codes is locale independent.

The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).

The compiler does not check if the value of a multibyte character or an escape sequence is a valid ISO/IEC 10646 code point. Such a value is encoded like the values of the valid members of the execution character set, according to the kind of the string literal (character or wide character).

The encoding of any of `wchar_t`, `char16_t`, and `char32_t` where the corresponding standard encoding macro (`__STDC_ISO_10646__`, `__STDC_UTF_16__`, or `__STDC_UTF_32__`) is not defined (6.10.8.2).

The symbol `__STDC_ISO_10646__` is not defined. Nevertheless every character in the Unicode required set, when stored in an object of type **`wchar_t`**, has the same value as the short identifier of that character.

The symbols `__STDC_UTF_16__` and `__STDC_UTF_32__` are defined.

7.7 Integers

Describes implementation-defined aspects of the ARM C compiler and C library relating to integers, as required by the ISO C standard.

Any extended integer types that exist in the implementation (6.2.5).

No extended integer types exist in the implementation.

Whether signed integer types are represented using sign and magnitude, two's complement, or ones' complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).

Signed integer types are represented using two's complement with no padding bits. There is no extraordinary value.

The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

No extended integer types exist in the implementation.

The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

When converting an integer to a N-bit wide signed integer type and the value cannot be represented in the destination type, the representation of the source operand is truncated to N-bits and the resulting bit patters is interpreted a value of the destination type. No signal is raised.

The results of some bitwise operations on signed integers (6.5).

In the bitwise right shift `E1 >> E2`, if `E1` has a signed type and a negative value, the value of the result is the integral part of the quotient of `E1 / 2^E2`, except that shifting the value –1 yields result –1.

7.8 Floating-point

Describes implementation-defined aspects of the ARM C compiler and C library relating to floating-point operations, as required by the ISO C standard.

The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results (5.2.4.2.2).

Floating-point quantities are stored in IEEE format:

- **`float`** values are represented by IEEE single-precision values
- **`double`** and **`long double`** values are represented by IEEE double-precision values.

The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>` (5.2.4.2.2).

The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>` is unknown.

The rounding behaviors characterized by non-standard values of `FLT_ROUNDS` (5.2.4.2.2).

ARM Compiler does not define non-standard values for `FLT_ROUNDS`.

The evaluation methods characterized by non-standard negative values of `FLT_EVAL_METHOD` (5.2.4.2.2).

ARM Compiler does not define non-standard values for `FLT_EVAL_METHOD`.

The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

The direction of rounding when an integer is converted to a floating point number is "round to nearest even".

The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).

When a floating-point number is converted to a different floating-point type and the value is within the range of the destination type, but cannot be represented exactly, the rounding mode is "round to nearest even", by default.

How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

When a floating-point literal is converted to a floating-point value, the rounding mode is "round to nearest even".

Whether and how floating expressions are contracted when not disallowed by the `FP_CONTRACT` pragma (6.5).

If `-ffp-mode=fast`, `-ffast-math`, or `-ffp-contract=fast` options are in effect, a floating-point expression can be contracted.

The default state for the `FENV_ACCESS` pragma (7.6.1).

The default state of the `FENV_ACCESS` pragma is `OFF`. The state `ON` is not supported.

Additional floating-point exceptions, rounding classifications, and their macro names (7.6, 7.12), modes, environments, and the default state for the `FP_CONTRACT` pragma (7.12.2).

No additional floating-point exceptions, rounding classifications, modes, or environments are defined.

The default state of `FP_CONTRACT` pragma is `OFF`.

7.9 Arrays and pointers

Describes implementation-defined aspects of the ARM C compiler and C library relating to arrays and pointers, as required by the ISO C standard.

The result of converting a pointer to an integer or vice versa (6.3.2.3).

Converting a pointer to an integer type with smaller bit width discards the most significant bits of the pointer. Converting a pointer to an integer type with greater bit width zero-extends the pointer. Otherwise the bits of the representation are unchanged.

Converting an unsigned integer to pointer with a greater bit-width zero-extends the integer. Converting a signed integer to pointer with a greater bit-width sign-extends the integer. Otherwise the bits of the representation are unchanged.

The size of the result of subtracting two pointers to elements of the same array (6.5.6).

The size of the result of subtracting two pointers to elements of the same array is 4 bytes for ARM instruction set architectures up to and including ARMv6, ARMv7, and ARMv8 AArch32, and 8 bytes for ARMv8 AArch64.

7.10 Hints

Describes implementation-defined aspects of the ARM C compiler and C library relating to registers, as required by the ISO C standard.

The extent to which suggestions made by using the register storage-class specifier are effective (6.7.1).

The register storage-class specifier is ignored as a means to control how fast the access to an object is. For example, an object might be allocated in register or allocated in memory regardless of whether it is declared with register storage-class.

The extent to which suggestions made by using the inline function specifier are effective (6.7.4).

The inline function specifier is ignored as a means to control how fast the calls to the function are made. For example, a function might be inlined or not regardless of whether it is declared inline.

7.11 Structures, unions, enumerations, and bitfields

Describes implementation-defined aspects of the ARM C compiler and C library relating to structures, unions, enumerations, and bitfields, as required by the ISO C standard.

Whether a plain `int` bit-field is treated as a signed `int` bit-field or as an unsigned `int` bit-field (6.7.2, 6.7.2.1).

Plain **`int`** bitfields are signed.

Allowable bit-field types other than `_Bool`, signed `int`, and `unsigned int` (6.7.2.1).

Enumeration types, **`long`** and **`long long`** (signed and unsigned) are allowed as bitfield types.

Whether atomic types are permitted for bit-fields (6.7.2.1).

Atomic types are not permitted for bitfields.

Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).

A bitfield cannot straddle a storage-unit boundary.

The order of allocation of bit-fields within a unit (6.7.2.1).

Within a storage unit, successive bit-fields are allocated from low-order bits towards high-order bits when compiling for little-endian, or from the high-order bits towards low-order bits when compiling for big-endian.

The alignment of non-bit-field members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

The non-bitfield members of structures of a scalar type are aligned to their size. The non-bitfield members of an aggregate type are aligned to the maximum of the alignments of each top-level member.

The integer type compatible with each enumerated type (6.7.2.2).

An enumerated type is compatible with **`int`** or **`unsigned int`**. If both the signed and the unsigned integer types can represent the values of the enumerators, the unsigned variant is chosen. If a value of an enumerator cannot be represented with **`int`** or **`unsigned int`**, then **`long long`** or **`unsigned long long`** is used.

7.12 Qualifiers

Describes implementation-defined aspects of the ARM C compiler and C library relating to qualifiers, as required by the ISO C standard.

What constitutes an access to an object that has volatile-qualified type (6.7.3).

Modifications of an object that has a volatile qualified type constitutes an access to that object. Value computation of an lvalue expression with a volatile qualified type constitutes an access to the corresponding object, even when the value is discarded.

7.13 Preprocessing directives

Describes implementation-defined aspects of the ARM C compiler and C library relating to preprocessing directives, as required by the ISO C standard.

The locations within `#pragma` directives where header name preprocessing tokens are recognized (6.4, 6.4.7).

The compiler does not support pragmas, which refer to headers.

How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

In both forms of the `#include` directive, the character sequences are mapped to external header names.

Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

The value of a character constant in conditional inclusion expression is the same as the value of the same constant in the execution character set.

Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

Single-character constants in conditional inclusion expressions have non-negative values.

The places that are searched for an included `< >` delimited header, and how the places are specified or the header is identified (6.10.2).

If the character sequence begins with the `/` character, it is interpreted as an absolute path name. Otherwise, the character sequence is interpreted as a file path, relative to one of the following directories:

- The sequence of the directories, given via the `-I` command line option, in the command line order.
- The `include` subdirectory in the compiler installation directory.

How the named source file is searched for in an included `" "` delimited header (6.10.2).

If the character sequence begins with the `/` character, it is interpreted as an absolute path name.

Otherwise, the character sequence interpreted as a file path, relative to the parent directory of the source file, which contains the `#include` directive.

The method by which preprocessing tokens (possibly resulting from macro expansion) in a `#include` directive are combined into a header name (6.10.2).

After macro replacement, the sequence of preprocessing tokens should be in one of the following two forms:

- A single string literal. The escapes in the string are not processed and adjacent string literals are not concatenated. Then the rules for double-quoted includes apply.
- A sequence of preprocessing tokens, starting with <' and terminating with >. Sequences of whitespace characters, if any, are replaced by a single space. Then the rules for angle-bracketed includes apply.

The nesting limit for `#include` processing (6.10.2).

There is no limit to the nesting level of files included with `#include`.

Whether the # operator inserts a \ character before the \ character that begins a universal character name in a character constant or string literal (6.10.3.2).

A \ character is inserted even before the \ character that begins a universal character name.

The behavior on each recognized non-standard C `#pragma` directive (6.10.6).

For the behavior of each non-standard C `#pragma` directive, see *Chapter 5 Compiler-specific Pragmas* on page 5-165.

The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (6.10.8.1).

The date and time of the translation are always available on all supported platforms.

7.14 Library functions

Describes implementation-defined aspects of the ARM C compiler and C library relating to library functions, as required by the ISO C standard.

Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).

The ARM Compiler provides the ARM C Micro-library. For information about facilities, provided by this library, see *The ARM C Micro-library* in the *ARM® C and C++ Libraries and Floating-Point Support User Guide*.

The format of the diagnostic printed by the assert macro (7.2.1.1).

The assert macro prints a diagnostic in the format:

```
*** assertion failed: expression, filename, line number
```

The representation of the floating-points status flags stored by the `fegetexceptflag` function (7.6.2.2).

The `fegetexceptflag` function stores the floating-point status flags as a bit set as follows:

- Bit 0 (`0x01`) is for the Invalid Operation exception.
- Bit 1 (`0x02`) is for the Divide by Zero exception.
- Bit 2 (`0x04`) is for the Overflow exception.
- Bit 3 (`0x08`) is for the Underflow exception.
- Bit 4 (`0x10`) is for the Inexact Result exception.

Whether the feraiseexcept function raises the Inexact floating-point exception in addition to the Overflow or Underflow floating-point exception (7.6.2.3).

The `feraiseexcept` function does not raise by itself the Inexact floating-point exception when it raises either an Overflow or Underflow exception.

Strings other than `"C"` and `""` that can be passed as the second argument to the setlocale function (7.11.1.1).

What other strings can be passed as the second argument to the `setlocale` function depends on which `__use_X_ctype` symbol is imported (`__use_iso8859_ctype`, `__use_sjis_ctype`, or `__use_utf8_ctype`), and on user-defined locales.

The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 (7.12).

The types defined for `float_t` and `double_t` are float and double, respectively, for all the supported values of `FLT_EVAL_METHOD`.

Domain errors for the mathematics functions, other than those required by this International Standard (7.12.1).

The following functions return additional domain errors under the specified conditions (the function name refers to all the variants of the function. For example, the `acos` entry applies to `acos`, `ascof`, and `acosl` functions):

Function	Condition	Return value	Error
`acos(x)`	`abs(x) > 1`	`NaN`	`EDOM`
`asin(x)`	`abs(x) > 1`	`NaN`	`EDOM`
`cos(x)`	`X == Inf`	`NaN`	`EDOM`
`sin(x)`	`x == Inf`	`NaN`	`EDOM`
`tan(x)`	`x == Inf`	`NaN`	`EDOM`
`atanh(x)`	`abs(x) == 1`	`Inf`	`ERANGE`
`ilogb(x)`	`x == 0.0`	`-INT_MAX`	`EDOM`
`ilogb(x)`	`x == Inf`	`INT_MAX`	`EDOM`
`ilogb(x)`	`x == NaN`	`FP_ILOGBNAN`	`EDOM`
`log(x)`	`x < 0`	`NaN`	`EDOM`
`log(x)`	`x == 0`	`-Inf`	`ERANGE`
`log10(x)`	`x < 0`	`NaN`	`EDOM`
`log10(x)`	`x == 0`	`-Inf`	`ERANGE`
`log1p(x)`	`x < -1`	`NaN`	`EDOM`
`log1p(x)`	`x == -1`	`-Inf`	`ERANGE`
`log2(x)`	`x < 0`	`NaN`	`EDOM`
`log2(x)`	`x == 0`	`-Inf`	`ERANGE`
`logb(x)`	`x == 0`	`-Inf`	`EDOM`
`logb(x)`	`x == Inf`	`+Inf`	`EDOM`
`pow(x, y)`	`y < 0` and `x == +0` or `y` is even	`+Inf`	`ERANGE`
`pow(x, y)`	`y < 0` and `x == -0` and `y` is odd	`-Inf`	`ERANGE`
`pow(x, y)`	`y < 0` and `x == -0` and `y` is non-integer	`+Inf`	`ERANGE`
`pow(x,y)`	`x < 0` and `y` is non-integer	`NaN`	`EDOM`
`sqrt(x)`	`x < 0`	`NaN`	`EDOM`
`lgamma(x)`	`x <= 0`	`Inf`	`ERANGE`
`tgamma(x)`	`x < 0` and `x` is integer	`NaN`	`EDOM`
`tgamma(x)`	`x == 0`	`Inf`	`ERANGE`
`fmod(x,y)`	`x == Inf`	`NaN`	`EDOM`
`fmod(x,y)`	`y == 0`	`NaN`	`EDOM`
`remainder(x, y)`	`y == 0`	`NaN`	`EDOM`
`remquo(x, y, q)`	`y == 0`	`NaN`	`EDOM`

The values returned by the mathematics functions on domain errors or pole errors (7.12.1).

See previous table.

The values returned by the mathematics functions on underflow range errors, whether `errno` is set to the value of the macro `ERANGE` when the integer expression `math_errhandling & MATH_ERRNO` is nonzero, and whether the Underflow floating-point exception is raised when the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero. (7.12.1).

On underflow, the mathematics functions return `0.0`, the `errno` is set to `ERANGE`, and the Underflow and Inexact exceptions are raised.

Whether a domain error occurs or zero is returned when an fmod function has a second argument of zero (7.12.10.1).

When the second argument of `fmod` is zero, a domain error occurs.

Whether a domain error occurs or zero is returned when a remainder function has a second argument of zero (7.12.10.2).

When the second argument of the remainder function is zero, a domain error occurs and the function returns NaN.

The base-2 logarithm of the modulus used by the remquo functions in reducing the quotient (7.12.10.3).

The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient is 4.

Whether a domain error occurs or zero is returned when a remquo function has a second argument of zero (7.12.10.3).

When the second argument of the `remquo` function is zero, a domain error occurs.

Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

The equivalent of `signal(sig, SIG_DFL)` is executed before the call to a signal handler.

The null pointer constant to which the macro NULL expands (7.19).

The macro `NULL` expands to 0.

Whether the last line of a text stream requires a terminating new-line character (7.21.2).

The last line of text stream does not require a terminating new-line character.

Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.21.2).

Space characters, written out to a text stream immediately before a new-line character, appear when read back.

The number of null characters that may be appended to data written to a binary stream (7.21.2).

No null characters are appended at the end of a binary stream.

Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.21.3).

The file position indicator of an append-mode stream is positioned initially at the end of the file.

Whether a write on a text stream causes the associated file to be truncated beyond that point (7.21.3).

A write to a text stream causes the associated file to be truncated beyond the point where the write occurred if this is the behavior of the device category of the file.

The characteristics of file buffering (7.21.3).

The C Library supports unbuffered, fully buffered, and line buffered streams.

Whether a zero-length file actually exists (7.21.3).

A zero-length file exists, even if no characters are written by an output stream.

The rules for composing valid file names (7.21.3).

Valid file names depend on the execution environment.

Whether the same file can be simultaneously open multiple times (7.21.3).

A file can be opened many times for reading, but only once for writing or updating.

The nature and choice of encodings used for multibyte characters in files (7.21.3).

The character input and output functions on wide-oriented streams interpret the multibyte characters in the associated files according to the current chosen locale.

The effect of the remove function on an open file (7.21.4.1).

Depends on the environment.

The effect if a file with the new name exists prior to a call to the rename function (7.21.4.2).

Depends on the environment.

Whether an open temporary file is removed upon abnormal program termination (7.21.4.3).

Depends on the environment.

Which changes of mode are permitted (if any), and under what circumstances (7.21.5.4)

No changes of mode are permitted.

The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.21.6.1, 7.29.2.1).

A double argument to the `printf` family of functions, representing an infinity is converted to `[-]inf`. A double argument representing a NaN is converted to `[-]nan`. The F conversion specifier, produces `[-]INF` or `[-]NAN`, respectively.

The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.21.6.1, 7.29.2.1).

The `fprintf` and `fwprintf` functions print %p arguments in lowercase hexadecimal format as if a precision of 8 (16 for 64-bit) had been specified. If the variant form (%#p) is used, the number is preceded by the character `@`.

——— **Note** ———

Using the `#` character with the `p` format specifier is undefined behavior in C11. `armclang` issues a warning.

———————————

The interpretation of a - character that is neither the first nor the last character, nor the second where a ^ character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.1).

`fscanf` and `fwscanf` always treat the character - in a `%...[...]` argument as a literal character.

The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.2).

`fscanf` and `fwscanf` treat `%p` arguments exactly the same as `%x` arguments.

The value to which the macro errno is set by the fgetpos, fsetpos, or ftell functions on failure (7.21.9.1, 7.21.9.3, 7.21.9.4).

On failure, the functions `fgetpos`, `fsetpos`, and `ftell` set the `errno` to `EDOM`.

The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function (7.22.1.3, 7.29.4.1.1).

Any n-char or n-wchar sequence in a string, representing a NaN, that is converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` functions, is ignored.

Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function sets errno to `ERANGE` when underflow occurs (7.22.1.3, 7.29.4.1.1).

The `strtod`, `strtold`, `wcstod`, `wcstof`, or `wcstold` functions set `errno` to `ERANGE` when underflow occurs.

The strtof function sets the errno to `ERANGE` by default (equivalent to compiling with `-ffp-mode=std`) and doesn't, when compiling with `-ffp-mode=full` or `-fno-fast-math`.

Whether the `calloc`, `malloc`, and `realloc` functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.22.3).

If the size of area requested is zero, `malloc()` and `calloc()` return a pointer to a zero-size block.

If the size of area requested is zero, `realloc()` returns `NULL`.

Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the abort or `_Exit` function is called (7.22.4.1, 7.22.4.5).

The function `_Exit` flushes the streams, closes all open files, and removes the temporary files.

The function `abort()` does not flush the streams and does not remove temporary files.

The termination status returned to the host environment by the `abort`, `exit`, `_Exit()`, or `quick_exit` function (7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7).

The function `abort()` returns termination status 1 to the host environment. The functions `exit()` and `_Exit()` return the same value as the argument that was passed to them.

The value returned by the system function when its argument is not a null pointer (7.22.4.8).

The value returned by the system function when its argument is not a null pointer depends on the environment.

The range and precision of times representable in `clock_t` and `time_t` (7.27).

The types `clock_t` and `time_t` can represent integers in the range [0, 4294967295].

The local time zone and Daylight Saving Time (7.27.1).

Depends on the environment.

The era for the clock function (7.27.2.1).

Depends on the environment.

The `TIME_UTC` epoch (7.27.2.5).

`TIME_UTC` and `timespec_get` are not implemented.

The replacement string for the `%Z` specifier to the `strftime` and `wcsftime` functions in the `"C"` locale (7.27.3.5, 7.29.5.1).

The functions `strftime` and `wcsftime` replace `%Z` with an empty string.

Whether the functions in `<math.h>` honor the rounding direction mode in an IEC 60559 conformant implementation, unless explicitly specified otherwise (F.10).

ARM Compiler does not declare `__STDC_IEC_559__` and does not support Annex F of ISO/IEC 9899:2011.

Related information

The ARM C and C++ Libraries.

7.15 Architecture

Describes implementation-defined aspects of the ARM C compiler and C library relating to architecture, as required by the ISO C standard.

The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (5.2.4.2, 7.20.2, 7.20.3).

——— Note ———

If the value column is empty, this means no value is assigned to the corresponding macro.

———————————

The values of the macros in `<float.h>` are:

Macro name	Value
FLT_ROUNDS	1
FLT_EVAL_METHOD	0
FLT_HAS_SUBNORM	
DBL_HAS_SUBNORM	
LDBL_HAS_SUBNORM	
FLT_RADIX	2
FLT_MANT_DIG	24
DBL_MANT_DIG	53
LDBL_MANT_DIG	53
FLT_DECIMAL_DIG	
DBL_DECIMAL_DIG	
LDBL_DECIMAL_DIG	
DECIMAL_DIG	17
FLT_DIG	6
DBL_DIG	15
LDBL_DIG	15
FLT_MIN_EXP	(-125)
DBL_MIN_EXP	(-1021)
LDBL_MIN_EXP	(-1021)
FLT_MIN_10_EXP	(-37)
DBL_MIN_10_EXP	(-307)
LDBL_MIN_10_EXP	(-307)
FLT_MAX_EXP	128
DBL_MAX_EXP	1024
LDBL_MAX_EXP	1024
FLT_MAX_10_EXP	38
DBL_MAX_10_EXP	308
LDBL_MAX_10_EXP	308
FLT_MAX	3.40282347e+38F
DBL_MAX	1.79769313486231571e+308
LDBL_MAX	1.79769313486231571e+308L

(continued)

Macro name	Value
FLT_EPSILON	1.19209290e-7F
DBL_EPSILON	2.2204460492503131e-16
LDBL_EPSILON	2.2204460492503131e-16L
FLT_MIN	1.175494351e-38F
DBL_MIN	2.22507385850720138e-308
LDBL_MIN	2.22507385850720138e-308L
FLT_TRUE_MIN	
DBL_TRUE_MIN	
LDBL_TRUE_MIN	

The values of the macros in <limits.h> are:

Macro name	Value
CHAR_BIT	8
SCHAR_MIN	(-128)
SCHAR_MAX	127
UCHAR_MAX	255
CHAR_MIN	0
CHAR_MAX	255
MB_LEN_MAX	6
SHRT_MIN	(-0x8000)
SHRT_MAX	0x7fff
USHRT_MAX	65535
INT_MIN	(~0x7fffffff)
INT_MAX	0x7fffffff
UINT_MAX	0xffffffffU
LONG_MIN	(~0x7fffffffL)
LONG_MIN (64-bit)	(~0x7fffffffffffffffL)
LONG_MAX	0x7fffffffL
LONG_MAX (64-bit)	0x7fffffffffffffffL
ULONG_MAX	0xffffffffUL
ULONG_MAX (64-bit)	0xffffffffffffffffUL
LLONG_MIN	(~0x7fffffffffffffffLL)
LLONG_MAX	0x7fffffffffffffffLL
ULLONG_MAX	0xffffffffffffffffULL

The values of the macros in <stdint.h> are:

Macro name	Value
INT8_MIN	-128
INT8_MAX	127
UINT8_MAX	255
INT16_MIN	-32768
INT16_MAX	32767
UINT16_MAX	65535
INT32_MIN	(~0x7fffffff)
INT32_MAX	2147483647
UINT32_MAX	4294967295u
INT64_MIN	(~0x7fffffffffffffffLL)
INT32_MAX	2147483647
UINT32_MAX	4294967295u
INT64_MIN (64-bit)	(~0x7fffffffffffffffL)
INT64_MAX (64-bit)	(9223372036854775807L)
UINT64_MAX (64-bit)	(18446744073709551615uL)
INT_LEAST8_MIN	-128
INT_LEAST8_MAX	127
UINT_LEAST8_MAX	255
INT_LEAST16_MIN	-32768
INT_LEAST16_MAX	32767
UINT_LEAST16_MAX	65535
INT_LEAST32_MIN	(~0x7fffffff)
INT_LEAST32_MAX	2147483647
UINT_LEAST32_MAX	4294967295u
INT_LEAST64_MIN	(~0x7fffffffffffffffLL)
INT_LEAST64_MAX	(9223372036854775807LL)
UINT_LEAST64_MAX	(18446744073709551615uLL)
INT_LEAST64_MIN (64-bit)	(~0x7fffffffffffffffL)
INT_LEAST64_MAX (64-bit)	(9223372036854775807L)
UINT_LEAST64_MAX (64-bit)	(18446744073709551615uL)
INT_FAST8_MIN	(~0x7fffffff)
INT_FAST8_MAX	2147483647
UINT_FAST8_MAX	4294967295u
INT_FAST16_MIN	(~0x7fffffff)
INT_FAST16_MAX	2147483647
UINT_FAST16_MAX	4294967295u

(continued)

Macro name	Value
INT_FAST32_MIN	(~0x7fffffff)
INT_FAST32_MAX	2147483647
UINT_FAST32_MAX	4294967295u
INT_FAST64_MIN	(~0x7fffffffffffffffLL)
INT_FAST64_MAX	(9223372036854775807LL)
UINT_FAST64_MAX	(18446744073709551615uLL)
INT_FAST64_MIN (64-bit)	(~0x7fffffffffffffffL)
INT_FAST64_MAX (64-bit)	(9223372036854775807L)
UINT_FAST64_MAX (64-bit)	(18446744073709551615uL)
INTPTR_MIN	(~0x7fffffff)
INTPTR_MIN (64-bit)	(~0x7fffffffffffffffLL)
INTPTR_MAX	2147483647
INTPTR_MAX (64-bit)	(9223372036854775807LL)
UINTPTR_MAX	4294967295u
UINTPTR_MAX (64-bit)	(18446744073709551615uLL)
INTMAX_MIN	(~0x7fffffffffffffffll)
INTMAX_MAX	(92233720368547758071l)
UINTMAX_MAX	(18446744073709551615ull)
PTRDIFF_MIN	(~0x7fffffff)
PTRDIFF_MIN (64-bit)	(~0x7fffffffffffffffLL)
PTRDIFF_MAX	2147483647
PTRDIFF_MAX (64-bit)	(9223372036854775807LL)
SIG_ATOMIC_MIN	(~0x7fffffff)
SIG_ATOMIC_MAX	2147483647
SIZE_MAX	4294967295u
SIZE_MAX (64-bit)	(18446744073709551615uLL)
WCHAR_MIN	0
WCHAR_MAX	0xffffffffU
WINT_MIN	(~0x7fffffff)
WINT_MAX	2147483647

The result of attempting to indirectly access an object with automatic or thread storage duration from a thread other than the one with which it is associated (6.2.4).

Access to automatic or thread storage duration objects from a thread other than the one with which the object is associated proceeds normally.

The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).

Defined in the ARM EABI.

Whether any extended alignments are supported and the contexts in which they are supported, and valid alignment values other than those returned by an `_Alignof` expression for fundamental types, if any (6.2.8).

Alignments, including extended alignments, that are a power of 2 and less than or equal to `0x10000000`, are supported.

The value of the result of the `sizeof` and `_Alignof` operators (6.5.3.4).

Type	sizeof	_Alignof
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	4
`long` (64-bit)	8	8
`long long`	8	8
`float`	4	4
`double`	8	8
`long double`	8	8
`long double` (64-bit)	16	16