Arm® Compiler

Version 6.6

armclang Reference Guide

Non-Confidential

Issue

DUI0774_l_en

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	4 November 2016	Non-Confidential	Arm Compiler v6.6 Release
H	8 May 2017	Non-Confidential	Arm Compiler v6.6.1 Release
I	29 November 2017	Non-Confidential	Arm Compiler v6.6.2 Release
J	28 August 2019	Non-Confidential	Arm Compiler v6.6.3 Release
K	26 August 2020	Non-Confidential	Arm Compiler v6.6.4 Release
L	31 January 2023	Non-Confidential	Arm Compiler v6.6.5 Release

Proprietary Notice

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

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

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

Inclusive language commitment

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

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

Contents

List of Figures

List of Tables

1. Introduction

The Arm® Compiler 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.

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

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

See the Arm® Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.
⚠ Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
⚠ Warning	Requirements for the system. Not following these requirements might result in system failure or damage.
⚠ Danger	Requirements for the system. Not following these requirements will result in system failure or damage.

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

1.2 Other information

See the Arm website for other relevant information.

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

2. 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 various Arm-specific options. Extra 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.

2.1 Support level definitions

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

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

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

Identification in the documentation

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

Product features

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

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

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

Beta product features

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

Beta product features are identified with [BETA].

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

Alpha product features

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

Alpha product features are identified with [ALPHA].

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

Community features

Arm Compiler 6 is built on LLVM technology and preserves the functionality of that technology where possible. This means that there are more features available in Arm Compiler that are not listed in the documentation. These extra features are known as community features. For information on these community features, see the Clang Compiler User's Manual.

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

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

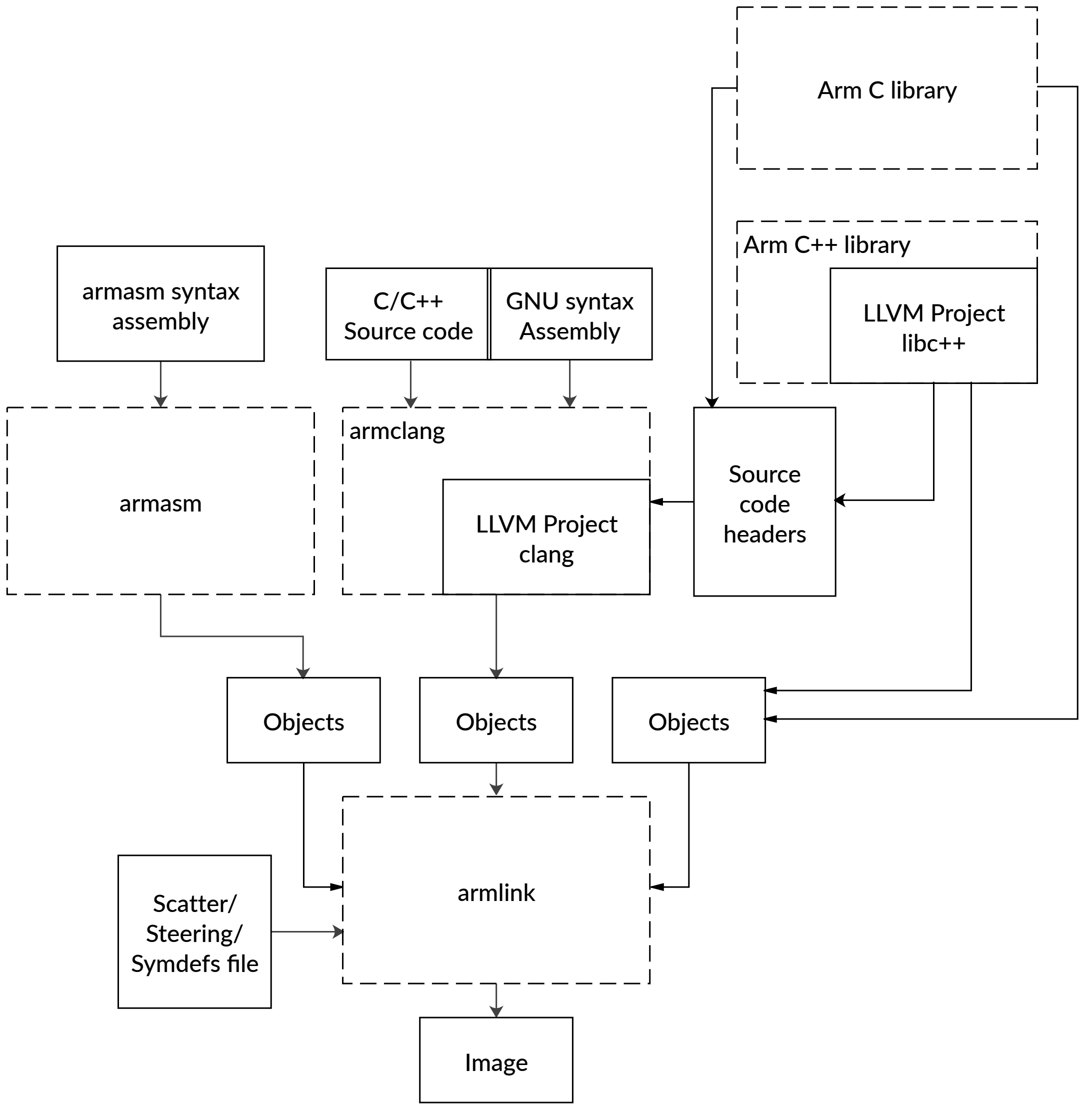
Some community features might become product features in the future, but Arm provides no roadmap for such features. Arm is interested in understanding your use of these features, and welcomes feedback on them. Arm supports customers using these features on a best-effort basis, unless the features are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

Guidance on use of community features

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

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

Figure 2-1: Integration boundaries in Arm Compiler for Embedded 6.



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

(ABI). Community features that do not span integration boundaries are more likely to work as expected.

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

Deprecated features

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

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

Unsupported features

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

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

List of known unsupported features

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

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

Note

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

- Use of C11 library features is unsupported.
- Any community feature that is exclusively related to non-Arm architectures is not supported.
- Except for Armv6-M, compilation for targets that implement architectures lower than Armv7 is not supported.
- The `long double` data type is not supported for AArch64 state because of limitations in the current Arm C library.

- C complex arithmetic is not supported, because of limitations in the current Arm C library.
- Complex numbers are defined in C++ as a template, `std::complex`. Arm Compiler supports `std::complex` with the `float` and `double` types, but not the `long double` type because of limitations in the current Arm C library.

Note

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

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

Warning

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

Alternatives to C complex numbers not being supported

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

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

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

2.2 Summary of armclang command-line options

This provides a summary of the `armclang` command-line options that Arm® Compiler 6 supports.

The command-line options either affect both compilation and assembly, or only affect compilation. The command-line options that only affect compilation without affecting armclang integrated assembler are shown in the table as *Compilation only*. The command-line options that affect both compilation and assembly are shown in the table as *Compilation and assembly*.

Note

The command-line options that affect assembly are for the `armclang` integrated assembler, and do not apply to armasm. These options affect both inline assembly and assembly language source files.

Note

Assembly language source files are assembled using the `armclang` integrated assembler. C and C++ language source files, which can contain inline assembly code, are compiled using the `armclang` compiler. Command-line options that are shown as *Compilation only* do not affect the integrated assembler, but they can affect inline assembly code.

Table 2-1: armclang command-line options

Option	Description	Compilation or Assembly
`-c`	Only perform the compile step, do not invoke `armlink`.	Compilation and assembly.
`-D`	Defines a preprocessor macro.	Compilation and assembly.
`-E`	Only perform the preprocess step, do not compile or link.	Compilation and assembly.
`-e`	Specifies the unique initial entry point of the image.	Compilation and assembly.
`-fbare-metal-pie`	Generates position-independent code.	Compilation only.
`-faggressive-jump-threading`, `-fno-aggressive-jump-threading`	Enables or disables the *Aggressive Jump Threading* (AJT) optimization.	Compilation only.
`-fbracket-depth`	Sets the limit for nested parentheses, brackets, and braces.	Compilation and assembly.
`-fcommon`, `-fno-common`	Generates common zero-initialized values for tentative definitions.	Compilation only.
`-fdata-sections`, `-fno-data-sections`	Enables or disables the generation of one ELF section for each variable in the source file.	Compilation only.
`-ffast-math`, `-fno-fast-math`	Enables or disables the use of aggressive floating-point optimizations.	Compilation only.
`-ffp-mode`	Specifies floating-point standard conformance.	Compilation only.
`-ffunction-sections`, `-fno-function-sections`	Enables or disables the generation of one ELF section for each function in the source file.	Compilation only.
`@file`	Reads a list of command-line options from a file.	Compilation and assembly.
`-fldm-stm`, `-fno-ldm-stm`	Enable or disable the generation of `LDM` and `STM` instructions. AArch32 only.	Compilation only.
`-fno-inline-functions`	Disables the automatic inlining of functions at optimization levels `-O2` and `-O3`.	Compilation only.
`-flto`	Enables link time optimization, and outputs bitcode wrapped in an ELF file for link time optimization.	Compilation only.

Option	Description	Compilation or Assembly
`-fexceptions`, `-fno-exceptions`	Enables or disables the generation of code needed to support C++ exceptions.	Compilation only.
`-fomit-frame-pointer`, `-fno-omit-frame-pointer`	Enables or disables the storage of stack frame pointers during function calls.	Compilation only.
`-fno-builtin`	Disables special handling and optimizations of standard C library functions.	Compilation only.
`-fropi`, `-fno-ropi`	Enables or disables the generation of *Read-Only Position-Independent* (ROPI) code.	Compilation only.
`-fropi-lowering`, `-fno-ropi-lowering`	Enables or disables runtime static initialization when generating ROPI code.	Compilation only.
`-frwpi`, `-fno-rwpi`	Enables or disables the generation of *Read-Write Position-Independent* (RWPI) code.	Compilation only.
`-frwpi-lowering`, `-fno-rwpi-lowering`	Enables or disables runtime static initialization when generating RWPI code.	Compilation only.
`-fshort-enums`, `-fno-short-enums`	Allows or disallows the compiler to set the size of an enumeration type to the smallest data type that can hold all enumerator values.	Compilation only.
`-fshort-wchar`, `-fno-short-wchar`	Sets the size of `wchar_t` to 2 or 4 bytes.	Compilation only.
`-fstack-protector, -fstack-protector-strong, -fstack-protector-all, -fno-stack-protector`	Inserts a guard variable onto the stack frame for each vulnerable function or for all functions.	Compilation only.
`-fstrict-aliasing`, `-fno-strict-aliasing`	Instructs the compiler to apply or not apply the strictest aliasing rules available.	Compilation only.
`-fvectorize`, `-fno-vectorize`	Enables or disables the generation of Advanced SIMD vector instructions directly from C or C++ code at optimization levels `-O1` and higher.	Compilation only.
`-ftrapv`	Instructs the compiler to generate traps for signed arithmetic overflow on addition, subtraction, and multiplication operations.	Compilation only.
`-fwrapv`	Instructs the compiler to assume that signed arithmetic overflow of addition, subtraction, and multiplication, wraps using two's-complement representation.	Compilation only.

Option	Description	Compilation or Assembly
`-g`, `-gdwarf-2`, `-gdwarf-3`, `-gdwarf-4`	Adds debug tables for source-level debugging.	Compilation and assembly.
`-I`	Adds the specified directory to the list of places that are searched to find include files.	Compilation and assembly.
`-include`	Includes the source code of the specified file at the beginning of the compilation.	Compilation only.
`-L`	Specifies a list of paths that the linker searches for user libraries.	Compilation only.
`-l`	Add the specified library to the list of searched libraries.	Compilation only.
`-M`, `-MM`	Produces a list of makefile dependency rules suitable for use by a make utility.	Compilation and assembly.
`-MD`, `-MMD`	Compiles or assembles source files and produces a list of makefile dependency rules suitable for use by a make utility.	Compilation and assembly.
`-MF`	Specifies a filename for the makefile dependency rules produced by the `-M` and `-MD` options.	Compilation only.
`-MG`	Prints dependency lines for header files even if the header files are missing.	Compilation only.
`-MP`	Emits dummy dependency rules that work around make errors that are generated if you remove header files without a corresponding update to the makefile.	Compilation only.
`-MT`	Changes the target of the makefile dependency rule produced by dependency generating options.	Compilation and assembly.
`-march`	Targets an architecture profile, generating generic code that runs on any processor of that architecture.	Compilation and assembly.
`-marm`	Requests that the compiler targets the A32 instruction set.	Compilation only.
`-mbig-endian`	Generates code suitable for an Arm processor using byte-invariant big-endian (BE-8) data.	Compilation and assembly.
`-mcmse`	Enables the generation of code for the Secure state of the Armv8-M Security Extensions.	Compilation only.
`-mcpu`	Targets a specific processor, generating optimized code for that specific processor.	Compilation and assembly.
`-mexecute-only`	Generates execute-only code, and prevents the compiler from generating any data accesses to code sections.	Compilation only.

Option	Description	Compilation or Assembly
`-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.	Compilation and assembly.
`-mfpu`	Specifies the target FPU architecture, that is the floating-point hardware available on the target.	Compilation and assembly.
`-mimplicit-it`	Specifies the behavior of the integrated assembler if there are conditional instructions outside `IT` blocks.	Compilation and assembly.
`-mlittle-endian`	Generates code suitable for an Arm processor using little-endian data.	Compilation and assembly.
`-munaligned-access`, `-mno-unaligned-access`	Enables or disables unaligned accesses to data on Arm processors.	Compilation only.
`-mthumb`	Requests that the compiler targets the T32 instruction set.	Compilation only.
`-nostdlib`	Tells the compiler to not use the Arm standard C and C++ libraries.	Compilation only.
`-nostdlibinc`	Tells the compiler to exclude the Arm standard C and C++ library header files.	Compilation only.
`-o`	Specifies the name of the output file.	Compilation and assembly.
`-O`	Specifies the level of optimization to use when compiling source files.	Compilation only.
`-pedantic`	Generate warnings if code violates strict ISO C and ISO C++.	Compilation only.
`-pedantic-errors`	Generate errors if code violates strict ISO C and ISO C++.	Compilation only.
`-Rpass` [COMMUNITY]	Outputs remarks from the optimization passes made by `armclang`. You can output remarks for all optimizations, or remarks for a specific optimization.	Compilation only.
`-S`	Outputs the disassembly of the machine code generated by the compiler.	Compilation only.
`-save-temps`	Instructs the compiler to generate intermediate assembly files from the specified C/C++ file.	Compilation only.
`-std`	Specifies the language standard to compile for.	Compilation only.
`--target`	Generate code for the specified target triple.	Compilation and assembly.
`-U`	Removes any initial definition of the specified preprocessor macro.	Compilation only.
`-u`	Prevents the removal of a specified symbol if it is undefined.	Compilation and assembly.
`-v`	Displays the commands that invoke the compiler and sub-tools, such as `armlink`, and executes those commands.	Compilation and assembly.

Option	Description	Compilation or Assembly
`--version`	Displays the same information as `--vsn`.	Compilation and assembly.
`--version_number`	Displays the version of `armclang` you are using.	Compilation and assembly.
`--vsn`	Displays the version information and the license details.	Compilation and assembly.
`-W`	Controls diagnostics.	Compilation only.
`-Wl`	Specifies linker command-line options to pass to the linker when a link step is being performed after compilation.	Compilation only.
`-Xlinker`	Specifies linker command-line options to pass to the linker when a link step is being performed after compilation.	Compilation only.
`-x`	Specifies the language of source files.	Compilation and assembly.
`-###`	Displays the commands that invoke the compiler and sub-tools, such as `armlink`, without executing those commands.	Compilation and assembly.

2.3 -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
```

2.4 -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 information

-include on page 54
-U on page 86
-x on page 91
Preprocessing assembly code

2.5 -E

Executes the preprocessor step only.

Operation

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 information

--target on page 85

2.6 -e

Specifies the unique initial entry point of the image.

Operation

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

2.7 -faggressive-jump-threading, -fno-aggressive-jump-threading

Enables or disables the `Aggressive Jump Threading` (AJT) optimization.

Default

`-faggressive-jump-threading` is the default when compiling at optimization levels `-O3`, `-Ofast`, and `-Omax`.

`-fno-aggressive-jump-threading` is the default when compiling at optimization levels `-O0`, `-O1`, `-O2`, `-Os`, `-Oz`, and `-Omin`.

Operation

AJT is an optimization that the compiler runs in addition to other optimizations that the compiler can perform. AJT is an extension to LLVM's Jump Threading Pass. For example,it can potentially optimize code that contains a series of switch statements inside a loop.

2.8 -fbare-metal-pie

Generates position independent code.

Operation

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 information

Bare-metal Position Independent Executables

2.9 -fbracket-depth=N

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

Default

The default depth limit is 256.

Syntax

```
-fbracket-depth=N
```

Usage

You can increase the depth limit *N*.

Related information

Translation limits on page 158

2.10 -fcommon, -fno-common

Generates common zero-initialized values for tentative definitions.

Operation

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 symbol 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`.

2.11 -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 information

-ffunction-sections, -fno-function-sections on page 36
__attribute__((section("name"))) variable attribute on page 124

2.12 -ffast-math, -fno-fast-math

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

Operation

`-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. 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, specify `-ffp-mode=std`.

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 2-2: Floating-point library variants

`armclang` option	Floating-point library variant	Description
Default	`fz`	IEEE-compliant except that denormals are flushed to zero and no exceptions are supported.
`-ffast-math`	`fz`	IEEE-compliant with the following exceptions: • Denormals are flushed to zero and no exceptions are supported. • `armclang` performs aggressive floating-point optimizations that might cause a small loss of accuracy.

`armclang` option	Floating-point library variant	Description
`-fno-fast-math`	`g`	IEEE-compliant library with configurable rounding mode and support for all IEEE exceptions, and flushing to zero. This library is a software floating-point implementation, and can result in extra code size and lower performance.

Related information

-ffp-mode on page 34

2.13 -ffp-mode

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

Default

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

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 model 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. Various 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, except for floating-point exceptions, 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 the *Arm C and C++ Libraries and Floating-Point Support User Guide*.

Note

When using the `std` or `fast` modes, the binary representation of a floating-point number that cannot be represented exactly by its type can differ depending on whether it is evaluated by the compiler at compile time or generated at run time using one of the following string to floating-point conversion functions:

- `atof()`.
- `strtod()`.
- `strtof()`.
- `strtold()`.
- A member of the `scanf()` family of functions using a floating-point conversion specifier.

Table 2-3: Floating-point library variant selection

`armclang` option	Floating-point library variant	Description
`-ffp-mode=std`	`fz`	IEEE-compliant except that denormals are flushed to zero and no exceptions are supported.
`-ffp-mode=fast`	`fz`	IEEE-compliant with the following exceptions: • Denormals are flushed to zero and no exceptions are supported. • `armclang` performs aggressive floating-point optimizations that might cause a small loss of accuracy.
`-ffp-mode=full`	`g`	IEEE-compliant library with configurable rounding mode and support for all IEEE exceptions, and flushing to zero. This library is a software floating-point implementation, and can result in extra code size and lower performance.

2.14 -ffunction-sections, -fno-function-sections

`-ffunction-sections` generates a separate ELF section for each function in the source file. The unused section elimination feature of the linker can then remove unused functions at link time.

Default

The default is `-ffunction-sections`.

Operation

The output section for each function has the same name as the function that generates the section, but with a `.text.` prefix. To prevent each function being placed in separate sections, 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")))`.

Post-conditions

`-ffunction-sections` reduces the potential for sharing addresses, data, and string literals between functions. Therefore, there might be a slight increase in code size 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 information

__attribute__((section("name"))) function attribute on page 113
-fdata-sections, -fno-data-sections on page 31

2.15 @file

Reads a list of armclang 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
```

2.16 -fldm-stm, -fno-ldm-stm

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

Default

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

Usage

The `-fno-ldm-stm` option can reduce interrupt latency on 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.

2.17 -fno-builtin

Disables special handling and optimization of standard C library functions, for example for `printf()`, `strlen()`, and `malloc()`.

Default

`-fno-builtin` is disabled by default.

Operation

When compiling without `-fno-builtin`, the compiler can replace calls to certain standard C library functions with inline code or with calls to other library functions. Then, your re-implementations of the standard C library functions might not be used, and might be removed by the linker. *Run-time ABI for the Arm Architecture* lists the library functions available.

Example

This example shows the result of compiling the following program with and without `-fno-builtin`:

```
#include "stdio.h"

void foo( void )
{
    printf("Hello\n");
}
```

1. Compile without `-fno-builtin`:

   ```
   armclang -c -O2 -g --target=arm-arm-none-eabi -mcpu=cortex-a9 -mfpu=none -
   nostdlib foo.c -o foo.o
   ```

2. Run the following `fromelf` command to show the disassembled output:

   ```
   fromelf --disassemble foo.o -o foo.s
   ...
   ||foo|| PROC
           ADR      r0,|L3.8|
           B        puts
   |L3.8|
           DCD      0x6c6c6548
           DCD      0x0000006f
           ENDP
   ...
   ```

   Note

   The compiler has replaced the `printf()` function with the `puts()` function.

3. Compile with `-fno-builtin`:

   ```
   armclang -c -O2 -g --target=arm-arm-none-eabi -mcpu=cortex-a9 -mfpu=none -
   nostdlib -fno-builtin foo.c -o foo.o
   ```

4. Run the following `fromelf` command to show the disassembled output:

```
fromelf --disassemble foo.o -o foo.s
...
||foo|| PROC
        ADR      r0,|L3.8|
        B        printf
|L3.8|
        DCD      0x6c6c6548
        DCD      0x00000a6f
        ENDP
...
```

Note

The compiler has not replaced the `printf()` function with the `puts()` function when using the `-fno-builtin` option.

Related information

-nostdlib on page 75
-nostdlibinc on page 76

2.18 -fno-inline-functions

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

Operation

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 information

-O (armclang) on page 78
Inline functions on page 149

2.19 -flto, -fno-lto

Enables or disables link time optimization. `-flto` outputs bitcode wrapped in an ELF file for link time optimization.

Default

The default is `-fno-lto`, except when you specify the optimization level `-Omax`.

Usage

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.

The compiler creates one file for each source file, with a `.o` file extension replacing the file extension on theinput 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` option `-Omax`.

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 by analyzing the dependencies between bitcode format objects. This can result in the removal of unused variables and functions in the source code.

Note

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

Related information

-c on page 27

2.20 -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

2.21 -fomit-frame-pointer, -fno-omit-frame-pointer

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

Default

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

Operation

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 A32 code or register `R7` for T32 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:

- By default, 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 A32 and T32 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.

2.22 -fropi, -fno-ropi

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

Default

The default is `-fno-ropi`.

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.

Restrictions

The following restrictions apply:

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

Related information

-frwpi, -fno-rwpi on page 44
-frwpi-lowering, -fno-rwpi-lowering on page 45
-fropi-lowering, -fno-ropi-lowering on page 43

2.23 -fropi-lowering, -fno-ropi-lowering

Enables or 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`.

2.24 -frwpi, -fno-rwpi

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

Default

The default is `-fno-rwpi`.

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 state.
- This option is not compatible with `-fpic`, `-fpie`, or `-fbare-metal-pie` options.

Related information

-fropi, -fno-ropi on page 43
-fropi-lowering, -fno-ropi-lowering on page 43

-frwpi-lowering, -fno-rwpi-lowering on page 45

2.25 -frwpi-lowering, -fno-rwpi-lowering

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

Default

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

Operation

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.

2.26 -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.

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.

Operation

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.

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)

2.27 -fshort-wchar, -fno-short-wchar

`-fshort-wchar` sets the size of wchar_t to 2 bytes. `-fno-short-wchar` sets the size of `wchar_t` to 4 bytes.

Default

The default is `-fno-short-wchar`.

Operation

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`.

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
```

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

Inserts a guard variable onto the stack frame for each vulnerable function or for all functions.

Default

The default is `-fno-stack-protector`.

Syntax

`-fstack-protector`

`-fstack-protector-all`

`-fstack-protector-strong`

`-fno-stack-protector`

Parameters

None

Operation

The prologue of a function stores a guard variable onto the stack frame. Before returning from the function, the function epilogue checks the guard variable to make sure that it has not been overwritten. A guard variable that is overwritten indicates a buffer overflow, and the checking code alerts the run-time environment.

`-fno-stack-protector` disables stack protection.

`-fstack-protector` enables stack protection for vulnerable functions that contain:

- A character array larger than 8 bytes.
- An 8-bit integer array larger than 8 bytes.
- A call to `alloca()` with either a variable size or a constant size bigger than 8 bytes.

`-fstack-protector-all` adds stack protection to all functions regardless of their vulnerability.

`-fstack-protector-strong` enables stack protection for vulnerable functions that contain:

- An array of any size and type.
- A call to `alloca()`.
- A local variable that has its address taken.

Note

If you specify more than one of these options, the last option that is specified takes effect.

When a vulnerable function is called with stack protection enabled, the initial value of its guard variable is taken from a global variable:

```
void *__stack_chk_guard;
```

You must provide this variable with a suitable value. For example, a suitable implementation might set this variable to a random value when the program is loaded, and before the first protected function is entered. The value must remain unchanged during the life of the program.

When the checking code detects that the guard variable on the stack has been modified, it notifies the run-time environment by calling the function:

```
void __stack_chk_fail(void);
```

You must provide a suitable implementation for this function. Normally, such a function terminates the program, possibly after reporting a fault.

Optimizations can affect the stack protection. The following are simple examples:

- Inlining can affect whether a function is protected.
- Removal of an unused variable can prevent a function from being protected.

Example: Stack protection

Create the following `main.c` and `get.c` files:

```
// main.c
#include <stdio.h>
#include <stdlib.h>

void *__stack_chk_guard = (void *)0xdeadbeef;

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

void get_input(char *data);

int main(void)
{
    char buffer[9];
    get_input(buffer);
    return buffer[0];
}
```

```
// get.c
```

```
#include <string.h>

void get_input(char *data)
{
    strcpy(data, "012345678");
}
```

When `main.c` and `get.c` are compiled with `-fstack-protector`, the array `buffer` is considered vulnerable and stack protection gets applied the function `main()`. The checking code recognizes the overflow of `buffer` that occurs in `get_input()`:

```
armclang --target=arm-arm-none-eabi -march=armv8-a -fstack-protector main.c get.c
```

Running the image displays the following message:

```
Stack smashing detected.
```

Related information

-Rpass on page 81

2.29 -fstrict-aliasing, -fno-strict-aliasing

Instructs the compiler to apply the strictest aliasing rules available.

Operation

`-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
```

2.30 -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
```

2.31 -fvectorize, -fno-vectorize

Enables or 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 information

-c on page 27
-O (armclang) on page 78

2.32 -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
```

2.33 -g, -gdwarf-2, -gdwarf-3, -gdwarf-4

Adds debug tables for source-level debugging.

Default

By default, `armclang` does not produce debug information. When using `-g`, the default level is DWARF 4.

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® Development Studio Debugger is compatible with DWARF 4. Compile with the `-g` or `-gdwarf-4` options to debug with Arm Development Studio 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 must only be used for backwards compatibility.

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`.

2.34 -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.

2.35 -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 information

-D on page 27
-I on page 53
-U on page 86

2.36 -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

2.37 -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

2.38 -M, -MM

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

`armclang` executes only the preprocessor step of the compilation or assembly. 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 or assemble 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 information

-o on page 77

2.39 -MD, -MMD

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

Operation

`armclang` creates a makefile dependency file for each source file, using a `.d` suffix. Unlike `-M` and `-MM`, that cause compilation or assembly to stop after the preprocessing stage, `-MD` and `-MMD` allow for compilation or assembly 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 information

-M, -MM on page 55
-MF on page 57

-MT on page 59

2.40 -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 information

-M, -MM on page 55
-MD, -MMD on page 56
-MT on page 59

2.41 -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
```

2.42 -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:
```

2.43 -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.

Operation

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 information

-M, -MM on page 55
-MD, -MMD on page 56
-MF on page 57

2.44 -march

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

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 Arm®v8-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).

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 microcontroller architecture profile without the Main Extension. Derived from the Armv6-M architecture. Only valid with `--target=arm-arm-none-eabi`.

`armv8-m.main`
: Armv8 microcontroller architecture profile with the Main Extension. 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`.

Note

There are no software floating-point libraries for AArch64 targets. At link time `armlink` links against AArch64 library code that can use floating-point and SIMD instructions and registers. This still applies if you compile the source with `–march=`*`name`*`+nofp+nosimd` to prevent the compiler from using floating-point and SIMD instructions and registers.

To prevent the use of any floating-point instruction or register, either re-implement the library functions or create your own library that does not use floating-point instructions or registers.

Related information

-mcpu on page 65
-marm on page 62
-mthumb on page 74
--target on page 85

2.45 -marm

Requests that the compiler targets the A32 instruction set.

Default

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

Operation

Most Arm®v7-A (and earlier) processors support two instruction sets. These are the A32 instruction set (formerly ARM), and the T32 instruction set (formerly Thumb®). Armv8-A processors in AArch32 state continue to support these two instruction sets, but with extra instructions. The Armv8-A processors also provide 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 and T32 instructions.
- M-profile processors execute T32 instructions.

Note

This option is only valid for targets that support the A32 instruction set. For example, the `-marm` option is not valid for targets in AArch64 state. The compiler ignores the `-marm` option and generates a warning when compiling for a target in AArch64 state.

Related information

-mthumb on page 74
--target on page 85
-mcpu on page 65

2.46 -mbig-endian

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

Default

The default is `-mlittle-endian`.

Related information

-mlittle-endian on page 73

2.47 -mcmse

Enables the generation of code for the Secure state of the Arm®v8-M Security Extension. This option is required when creating a Secure image.

Note

- The Armv8-M Security Extension is not supported when building *Read-Only Position-Independent* (ROPI) and *Read-Write Position-Independent* (RWPI) images.
- Mixing objects compiled for Armv8-M.baseline and Armv8-M.mainline, could potentially leak sensitive data, because Armv8-M.baseline does not support the Floating-Point Extension. Therefore, the compiler cannot generate code to clear the Secure floating-point registers when performing a Non-secure call. If any object is compiled for the Armv8-M.mainline architecture, all files containing CMSE attributes must be compiled for the Armv8-M.mainline architecture.

Usage

Specifying `-mcmse` targets the Secure state of the Armv8-M Security Extension. When compiling with `-mcmse`, 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-precision floating-point members 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 reference are also unsupported and produce an error.
- The following cases are not supported when compiling with `-mcmse` and produce 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 information

-march on page 60
-mfpu on page 70
--target on page 85
__attribute__((cmse_nonsecure_call)) function attribute on page 104
__attribute__((cmse_nonsecure_entry)) function attribute on page 105
Predefined macros on page 143
TT instruction intrinsics on page 152
Non-secure function pointer intrinsics on page 155
Building Secure and Non-secure Images Using Armv8-M Security Extension
TT, TTT, TTA, TTAT instruction
--fpu linker option
--import_cmse_lib_in linker option
--import_cmse_lib_out linker option
--scatter linker option

2.48 -mcpu

Enables code generation for a specific Arm® processor.

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=name -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`.

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.

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

Use `+feature` or `+nofeature` 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, which enables AES, SHA1, and SHA256.
- `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.
- To write code that generates instructions for these extensions, use the intrinsics described in the Arm C Language Extensions.
- Arm Compiler 6.6 does not support the optional extensions to Armv8.2-A:
  - SHA3, SHA512, SM3, and SM4 cryptographic instructions.
  - SDOT, UDOT, VSDOT, and VUDOT dot product instructions.
  - Half-precision floating-point multiply with add or multiply with subtract arithmetic instructions.

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 are no software floating-point libraries for AArch64 targets. When linking for AArch64 targets, `armlink` uses AArch64 libraries that contain floating-point and Advanced SIMD instructions and registers. This applies even if you compile the source with `-mcpu=`*`name`*`+nofp+nosimd` to prevent the compiler from using floating-point and Advanced SIMD instructions and registers. Therefore, there is no guarantee that the linked image for AArch64 targets is entirely free of floating-point and Advanced SIMD instructions and registers.

You can prevent the use of floating-point and Advanced SIMD instructions and registers in images that are linked for AArch64 targets. To do this, re-implement the library functions or create your own library that does not use floating-point and Advanced SIMD instructions and registers.

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
```

To target the AArch32 state of a Cortex-A53 processor, generating T32 instructions:

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

Related information

-marm on page 62
-mthumb on page 74
--target on page 85
-mfpu on page 70

2.49 -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 T32 code.

Execute-only code is only supported for:

- Processors that support the Arm®v8-M architecture, with or without the Main Extension.
- 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

LTO 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

2.50 -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.

Default

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

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.

Note

In AArch32 state, if you specify `-mfloat-abi=soft`, then specifying the `-mfpu` option does not have an effect.

Related information

-mfpu on page 70

2.51 -mfpu

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

Default

The default FPU option depends on the target processor.

Syntax

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

```
-mfpu=list
```

Note

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

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 uses software-based floating-point library functions. This is similar to the `-mfloat-abi=soft` option.

`vfpv3`

Enable the Arm®v7 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 Arm® Neon® Advanced SIMD extension.

Note

There are no software floating-point libraries for AArch64 targets. When linking for AArch64 targets, `armlink` uses AArch64 libraries that contain floating-point and Advanced SIMD instructions and registers. This applies even if you compile the source with `-mcpu=`*`name`*`+nofp+nosimd` to prevent the compiler from using floating-point and Advanced SIMD instructions and registers. Therefore, there is no guarantee that the linked image for AArch64 targets is entirely free of floating-point and Advanced SIMD instructions and registers.

You can prevent the use of floating-point and Advanced SIMD instructions and registers in images that are linked for AArch64 targets. To do this, re-implement the library functions or create your own library that does not use floating-point and Advanced SIMD instructions and registers.

Note

In AArch32 state, if you specify `-mfloat-abi=soft`, then specifying the `-mfpu` option does not have an effect.

Related information

-mcpu on page 65
-mfloat-abi on page 69
--target on page 85

2.52 -mimplicit-it

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

Default

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

Syntax

`-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 for targets 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 with A64 code.

2.53 -mlittle-endian

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

Default

The default is `-mlittle-endian`.

Related information

-mbig-endian on page 63

2.54 -mthumb

Requests that the compiler targets the T32 instruction set.

Default

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

Operation

Most Arm®v7-A (and earlier) processors support two instruction sets. These are the A32 instruction set (formerly ARM), and the T32 instruction set (formerly Thumb®). Armv8-A processors in AArch32 state continue to support these two instruction sets, but with extra instructions. The Armv8-A processors also provide 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 and T32 instructions.
- M-profile processors execute T32 instructions.

Note

- The `-mthumb` option is not valid for targets in AArch64 state, for example `--target=aarch64-arm-none-eabi`. The compiler ignores the `-mthumb` option and generates a warning when compiling for a target in AArch64 state.
- 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 instruction set for your assembly source files, you must use the `.thumb` or `.code 16` directive in the assembly files.

Example

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

Related information

-marm on page 62
--target on page 85
-mcpu on page 65

2.55 -munaligned-access, -mno-unaligned-access

Enables or disables unaligned accesses to data on Arm processors.

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 without the Main Extension.

Operation

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.

`-munaligned-access`

Use this option on processors that support unaligned accesses to data, to speed up accesses to packed structures.

Note

Compiling with this option generates an error for the following architectures:

- Armv6-M.
- Armv8-M without the Main Extension.

`-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 information

Predefined macros on page 143
Arm C Language Extensions 2.0

2.56 -nostdlib

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

Default

`-nostdlib` is disabled by default.

Operation

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

The `armclang` option `-nostdlib`, 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 `armlink` option `--userlibpath`.

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 `armclang` option `-nostdlib`. Your libraries must be compliant with the ISO C library and with the AEABI specification.

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`, 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 the C Standard and the AEABI do not define.

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

Related information

-nostdlibinc on page 76
Run-time ABI for the Arm Architecture
C Library ABI for the Arm Architecture

2.57 -nostdlibinc

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

Note

This option still searches the `lib/clang/*/include` directory.

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

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
```

Related information

-nostdlib on page 75

2.58 -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 2-4: Compiling without the -o option

Compiler option	Action	Usage notes
`-c`	Produces an object file whose name defaults to *`filename`*`.o` in the current directory. *`filename`* is the name of the source file stripped of any leading directory names.	-
`-S`	Produces an assembly file whose name defaults to *`filename`*`.s` in the current directory. *`filename`* is the name of the source file stripped of any leading directory names.	-
`-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

2.59 -O (armclang)

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

Default

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

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 optimization might result in a significantly larger image.

1

Restricted optimization. When debugging is enabled, this option selects a good compromise between image size, performance, and quality of debug view.

Arm recommends `-O1` rather than `-O0` for the best trade-off between debug view, code size, and performance.

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 the debug information cannot describe.

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 optimizations that are performed with the `armclang` option `-ffp-mode=fast`. 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` option `-flto` 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. The reason is because the `-flto` option is turned on by default at `-Omax`, and that option 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

LTO does not honor the `armclang` option `-mexecute-only`. If you use the `armclang` options `-flto` or `-Omax`, 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.

Related information

-flto, -fno-lto on page 40
-fropi, -fno-ropi on page 43
-frwpi, -fno-rwpi on page 44
Restrictions with link time optimization

2.60 -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 a warning.

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

Note

The `-pedantic` option is stricter than the `-Wpedantic` option.

Related information

-std on page 84
-W on page 89
-pedantic-errors on page 80

2.61 -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 information

-std on page 84
-W on page 89
-pedantic on page 80

2.62 -Rpass

Outputs remarks from the optimization passes made by `armclang`. You can output remarks for all optimizations, or remarks for a specific optimization.

Note

This topic describes a [COMMUNITY] feature. See Support level definitions.

Syntax

`-Rpass={.*|`*`optimization`*`}`

`-Rpass-missed={.*|`*`optimization`*`}` [COMMUNITY]

Parameters

Where:

`.*`

Indicates that remarks for all major optimizations such as inlining, vectorization, and loop optimizations are to be reported. However, not all optimization passes support this feature.

`optimization`

Is a specific optimization for which remarks are to be output. See the Clang Compiler User's Manual for more information about the optimization values you can specify.

Example

The following examples use the file:

```
// test.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void *__stack_chk_guard = (void *)0xdeadbeef;

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

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

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

- To display the inlining remarks, specify:

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -O2 -Rpass=inline test.c
test.c:22:3: remark: 'copy' inlined into 'main' with (cost=-14980, threshold=337)
 at callsite main:2:3; [-Rpass=inline]
  copy(t);
  ^
```

- To display the stack protection remarks, specify:

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -O0 -fstack-protector -
Rpass=stack-protector test.c
test.c:20:5: remark: Stack protection applied to function main due to a stack
 allocated buffer or struct containing a
      buffer [-Rpass=stack-protector]
static void main(void) {
            ^
```

Related information

-fstack-protector, -fstack-protector-all, -fstack-protector-strong, -fno-stack-protector on page 47

2.63 -S

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

Operation

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 information

-o on page 77
-O (armclang) on page 78
-flto, -fno-lto on page 40

2.64 -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 information

-c on page 27
-S on page 83

2.65 -std

Specifies the language standard to compile for.

Note

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

Note

Arm® does not guarantee the compatibility of C++ compilation units compiled with different major or minor versions of Arm Compiler and linked into a single image. Therefore, Arm recommends that you always build your C++ code from source with a single version of the toolchain.

You can mix C++ with C code or C libraries.

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`

[COMMUNITY] C as defined by the 2011 C standard.

`gnu11`

[COMMUNITY] 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.

c++14 [BETA]
C++ as defined by the 2014 C++ standard.

gnu++14 [BETA]
C++ as defined by the 2014 C++ 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.

Note

`armclang` always applies the rules for type auto-deduction for copy-list-initialization and direct-list-initialization from C++17, regardless of which C++ source language mode a program is compiled for. For example, the compiler always deduces the type of `foo` as `int` instead of `std::initializer_list<int>` in the following code:

```
auto foo{ 1 };
```

Related information

Language Compatibility

2.66 --target

Generate code for the specified target triple.

Default

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

Syntax

`--target=`*`triple`*

Where:

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

Supported target triples 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 target triples 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.

Related information

-marm on page 62
-mthumb on page 74
-mcpu on page 65
-mfpu on page 70

2.67 -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 `-Uname` 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 `-Dname`.
3. Macros explicitly undefined, using `-Uname`.

Related information

-D on page 27
Predefined macros on page 143
-include on page 54

2.68 -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

2.69 -v

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

-### on page 92

2.70 --version

Displays the same information as `--vsn`.

Related information

--vsn on page 89

2.71 --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 information

Predefined macros on page 143

2.72 --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 --vsn
Product: ARM Compiler N.n.p
Component: ARM Compiler N.n.p
Tool: armclang [tool_id]

Target: target_name
```

Related information

--version on page 88
--version_number on page 88

2.73 -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.

`-Wfoo`
: 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 information

-pedantic-errors on page 80

-pedantic on page 80

2.74 -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 information

-Xlinker on page 91

2.75 -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 information

-Wl on page 90

2.76 -x

Specifies the language of source files.

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`.

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.

Related information

-D on page 27
Preprocessing assembly code

2.77 -###

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 information

-v on page 87

3. Compiler-specific Keywords and Operators

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

3.1 Keyword extensions

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 information

__alignof__ on page 94
__asm on page 96
__declspec attributes on page 97
__inline on page 100

3.2 __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
```

3.3 __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))` function attribute on the function declaration. For more information, see __attribute__((naked)) function attribute. 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 information

__attribute__((naked)) function attribute on page 110

3.4 __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 information

__declspec(noinline) on page 98

__declspec(noreturn) on page 98
__declspec(nothrow) on page 99

3.5 __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);
```

3.6 __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;
}
```

3.7 __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

3.8 __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 information

Inline functions on page 149

3.9 __promise

`__promise` represents a promise you make to the compiler that a given expression always has a nonzero value. This enables the compiler to perform more aggressive optimization when vectorizing code.

Syntax

`__promise(`*`expr`*`)`

Where *`expr`* is an expression that evaluates to nonzero.

Usage

You must `#include <assert.h>` to use `__promise(`*`expr`*`)` .

If assertions are enabled (by not defining `NDEBUG`) and the macro `__DO_NOT_LINK_PROMISE_WITH_ASSERT` is not defined, then the promise is checked at runtime by evaluating *`expr`* as part of `assert(`*`expr`*`)` .

3.10 __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 information

-munaligned-access, -mno-unaligned-access on page 74

4. 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.

4.1 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 4-1: 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__((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`*`")))`	-

Function attribute	Non-attribute equivalent
`__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 information

__attribute__((always_inline)) function attribute on page 103
__attribute__((const)) function attribute on page 105
__attribute__((constructor(priority))) function attribute on page 106
__attribute__((format_arg(string-index))) function attribute on page 107
__attribute__((malloc)) function attribute on page 109
__attribute__((nonnull)) function attribute on page 111
__attribute__((naked)) function attribute on page 110
__attribute__((pcs("calling_convention"))) function attribute on page 112
__attribute__((noinline)) function attribute on page 110
__attribute__((nothrow)) function attribute on page 112
__attribute__((section("name"))) function attribute on page 113
__attribute__((pure)) function attribute on page 113
__attribute__((noreturn)) function attribute on page 111
__attribute__((unused)) function attribute on page 114
__attribute__((used)) function attribute on page 113
__attribute__((visibility("visibility_type"))) function attribute on page 116
__attribute__((weak)) function attribute on page 117
__attribute__((weakref("target"))) function attribute on page 117
__alignof__ on page 94
__asm on page 96
__declspec attributes on page 97

4.2 __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
}
```

4.3 __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 information

__attribute__((cmse_nonsecure_entry)) function attribute on page 105
Non-secure function pointer intrinsics on page 155
Building Secure and Non-secure Images Using Armv8-M Security Extension

4.4 __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 information

__attribute__((cmse_nonsecure_call)) function attribute on page 104
Non-secure function pointer intrinsics on page 155
Building Secure and Non-secure Images Using Armv8-M Security Extension

4.5 __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);
    }
}
```

4.6 __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.

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()
```

4.7 __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.
```

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

This 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`.

Operation

This 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 Arm®v6-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 this attribute does not affect the behavior of the compiled output. However, Arm recommends using this attribute on exception handlers for clarity and easier software porting.

Note

- For architectures that support A32 and T32 instructions, functions specified with this attribute compile to A32 or T32 code depending on whether the compile option specifies A32 code or T32 code.
- For T32 only architectures, for example the Armv6-M architecture, functions specified with this attribute compile to T32 code.
- This attribute is not available for A64 code.
- When targeting the R-profile or A-profile, the *`type`* `UNDEF` can only handle **UNDEFINED** A32 instructions and **UNDEFINED** 2-byte sized T32 instructions. Assembly language is required to handle 4-byte sized T32 **UNDEFINED** instructions.

4.9 __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));
```

4.10 __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.

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 information

__asm on page 96

4.11 __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));
```

4.12 __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));
```

4.13 __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.

4.14 __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.

4.15 __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")));
```

4.16 __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.

4.17 __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.

4.18 __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, or use the `--no_remove` linker option to disable unused section elimination.

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

4.19 __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 information

__attribute__((unused)) variable attribute on page 125

4.20 __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);
```

4.21 __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()
{
   ...
}
```

4.22 __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));
```

4.23 __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.

4.24 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 information

__attribute__((aligned)) type attribute on page 118
__attribute__((transparent_union)) type attribute on page 119
__attribute__((packed)) type attribute on page 118

4.25 __attribute__((aligned)) type attribute

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

4.26 __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.

Examples

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

short get_y(struct foobar *s)
{
    // Correct usage: the compiler does 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 information

-munaligned-access, -mno-unaligned-access on page 74

4.27 __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 */
}
```

4.28 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 information

__attribute__((alias)) variable attribute on page 121
__attribute__((aligned)) variable attribute on page 122
__attribute__((deprecated)) variable attribute on page 122
__attribute__((packed)) variable attribute on page 123
__attribute__((section("name"))) variable attribute on page 124
__attribute__((unused)) variable attribute on page 125
__attribute__((used)) variable attribute on page 124
__attribute__((weak)) variable attribute on page 125
__attribute__((weakref("target"))) variable attribute on page 126

4.29 __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
}
```

4.30 __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));
```

4.31 __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.
```

4.32 __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 information

__attribute__((aligned)) variable attribute on page 122

4.33 __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 more 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.

4.34 __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
```

4.35 __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 information

__attribute__((unused)) function attribute on page 114

4.36 __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.

4.37 __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`.

5. 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>`.

5.1 __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 targets in AArch32 state.

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 A32 code

0 ... 255

if you are compiling source as T32 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);
    ...
}
```

5.2 __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 targets in AArch32 state.

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.

5.3 __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 targets in AArch32 state.

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.

5.4 __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 targets in AArch32 state.

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();
}
```

5.5 __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 targets in AArch32 state.

Note

Typically, this intrinsic disables IRQ interrupts by setting the I-bit in the CPSR. However, for Cortex®-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`.

5.6 __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 targets in AArch32 state.

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`.

5.7 __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 targets in AArch32 state.

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`.

5.8 __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 targets in AArch32 state.

This intrinsic also acts as a `__schedule_barrier` intrinsic.

Syntax

```
void __force_stores(void)
```

5.9 __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 targets in AArch32 state.

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)
```

5.10 __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 targets in AArch32 state.

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)
```

5.11 __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 targets in AArch32 state.

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 A32 state, excluding M-profile architectures.

`SVC 0xAB`

In T32 state, excluding M-profile architectures. This behavior is not guaranteed on all debug targets from Arm® or from third parties.

`HLT 0xF000`

In A32 state, excluding M-profile architectures.

`HLT 0x3C`

In T32 state, excluding M-profile architectures.

`BKPT 0xAB`

For M-profile architectures (T32 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 that 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

5.12 __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 targets in AArch32 state.

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 5-1: 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.

6. Compiler-specific Pragmas

Summarizes the Arm® Compiler-specific pragmas that are extensions to the C and C++ Standards.

6.1 #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.

6.2 #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 information

-W on page 89

6.3 #pragma clang section

Specifies names for one or more section types. The compiler places subsequent functions, global variables, or static variables in the named section depending on the section type. The names only apply within the 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 clang 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 clang 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.
```

6.4 #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.

6.5 #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 are `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.

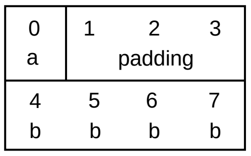
Example

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 };
```

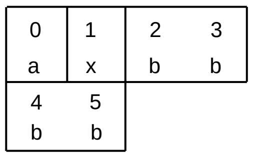
The layout of `S` is:

Figure 6-1: Nonpacked structure S



The layout of `SP` is:

Figure 6-2: 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`.

6.6 #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` does 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.

6.7 #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)
{
    ...
}
```

7. Other Compiler-specific Features

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

7.1 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 the Armv6 architecture.

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

Related information

Arm C Language Extensions

7.2 Predefined macros

The Arm® Compiler predefines various 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 Arm Compiler for C and C++, together with 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 7-1: 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 targets in AArch64 state only. Set to 1 if code is for 64-bit state.
`__ARM_ALIGN_MAX_STACK_PWR`	`4`	Set for targets in AArch64 state 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 targets in AArch32 state only. Set to 1 if hardware divide instructions are available.
`__ARM_ARCH_ISA_A64`	`1`	Set for targets in AArch64 state 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.

Name	Value	When defined
`__ARM_FEATURE_CMSE`	*num*	Indicates the availability of the Armv8-M Security Extension 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 Extension, but is closely related. `3` The Armv8-M Security Extension for secure executable files is available. This implies that the `TT` instruction is available. See TT instruction intrinsics 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 targets in AArch32 state 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 targets in AArch32 state only. Set to 1 if the Q (saturation) flag exists. **Note:** This macro is deprecated in ACLE 2.0 for A-profile.

Name	Value	When defined
`__ARM_FEATURE_SAT`	1	Set for targets in AArch32 state 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.
`__ARM_FEATURE_SIMD32`	1	Set for targets in AArch32 state 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 Arm® 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 targets in AArch32 state. Double-precision is always set for targets in AArch64 state.

Name	Value	When defined
`__ARM_PCS`	`1`	Set for targets in AArch32 state 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 targets in AArch32 state 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.
`__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 AArch32 state with `--target=arm-arm-none-eabi`. See also `__aarch64__`.
`__aarch64__`	`1`	See also `__arm__`.

Name	Value	When defined
`__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. An integer that specifies the major version of the compatible GCC version. This macro indicates that the compiler accepts GCC compatible code. The macro does not indicate whether the `-std` option has enabled GNU C extensions. For detailed Arm Compiler version information, use the `__ARMCOMPILER_VERSION` macro.
`__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	Defined depending on the `-mfloat-abi` value and whether a target has hardware floating-point support. See When the __SOFTFP__ predefined macro is 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`.

Name	Value	When defined
`__WINT_TYPE__`	*`type`*	Always set. Defines the correct underlying type for the `wint_t typedef`.

When the __SOFTFP__ predefined macro is defined

`__SOFTFP__` is defined as follows:

Table 7-2: __SOFTFP__ predefined macro

`-mfloat-abi=`*`value`*	Targets with hardware floating-point support	Targets without hardware floating-point support
Default	`__SOFTFP__` not defined	`__SOFTFP__` defined and set to 1
hard	`__SOFTFP__` not defined	`__SOFTFP__` not defined
soft	`__SOFTFP__` defined and set to 1	`__SOFTFP__` defined and set to 1
softfp	`__SOFTFP__` not defined	`__SOFTFP__` defined and set to 1

Related information

Compiler Command-line Options on page 18

7.3 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 information

__inline on page 100
-std on page 84
__attribute__((always_inline)) function attribute on page 103

7.4 Volatile variables

Arm® Compiler does not guarantee that a single-copy atomic instruction is used to access a volatile variable that is larger than the natural architecture data size, even when one is available for the target processor.

When compiling for AArch64 state, the natural architecture data size is 64-bits. Targets such as the Cortex®-A53 processor support single-copy atomic instructions for 128-bit data types. In this case, you might expect the compiler to generate an instruction with single-copy atomicity to access a `volatile` 128-bit variable. However, the architecture does not guarantee single-copy atomicity access. Therefore, the compiler does not support it.

When compiling for AArch32 state, the natural architecture data size is 32-bits. In this case, you might expect the compiler to generate an instruction with single-copy atomicity to access a `volatile` 64-bit variable. However, the architecture does not guarantee single-copy atomicity access. Therefore, the compiler does not support it.

Related information

Effect of the volatile keyword on compiler optimization
Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile

7.5 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 7-1: IEEE half-precision floating-point format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S			E							T					

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 Signaling:
                  0: Quiet NaN
                  1: Signaling 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

7.6 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 Extension

The following table describes the `TT` intrinsics for Arm®v8-M Security Extension 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.

Intrinsic	Description
`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 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.
`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 Extension

The semantics of the intrinsic `cmse_check_address_range()` are extended to handle the extra flag and fields introduced by the Armv8-M Security Extension.

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 information

Predefined macros on page 143

7.7 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 information

__attribute__((cmse_nonsecure_call)) function attribute on page 104
__attribute__((cmse_nonsecure_entry)) function attribute on page 105

Building Secure and Non-secure Images Using Armv8-M Security Extension

8. Standard C Implementation Definition

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

8.1 C Implementation definition

Appendix J of the ISO C standard (ISO/IEC 9899:2011 (E)) contains 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 information

Translation on page 157
Translation limits on page 158
Environment on page 159
Identifiers on page 160
Characters on page 161
Integers on page 163
Floating-point on page 163
Arrays and pointers on page 165
Hints on page 165
Structures, unions, enumerations, and bitfields on page 166
Qualifiers on page 166
C Preprocessing directives on page 167
Library functions on page 168
Architecture on page 173

8.2 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 `-W`*`flag`*, 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.

8.3 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 8-1: 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`*

Description	Translation limit
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 information

-fbracket-depth=N on page 30

8.4 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.

8.5 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.

8.6 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 10646.

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 `NULL` (`\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.

8.7 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.

8.8 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` values are represented by IEEE double-precision values.
- `long double` values in AArch32 are represented by IEEE double-precision values.
- `long double` values in AArch64 are represented by IEEE quadruple-precision values.

Note

The `long double` data type is not supported for AArch64 state because of limitations in the current Arm® C library.

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`.

8.9 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 AArch32 state, and 8 bytes for AArch64 state.

8.10 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.

8.11 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.

8.12 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.

8.13 C 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 that 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 file 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 file 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 must 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 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 Compiler-specific Pragmas.

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.

8.14 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`

Function	Condition	Return value	Error
`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 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 does not, 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

8.15 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` (AArch32)	`53`
`LDBL_MANT_DIG` (AArch64)	`113`
`FLT_DECIMAL_DIG`	-
`DBL_DECIMAL_DIG`	-
`LDBL_DECIMAL_DIG` (AArch32)	-
`LDBL_DECIMAL_DIG` (AArch64)	`36`
`DECIMAL_DIG`	`17`
`FLT_DIG`	`6`
`DBL_DIG`	`15`
`LDBL_DIG` (AArch32)	`15`
`LDBL_DIG` (AArch64)	`33`
`FLT_MIN_EXP`	`(-125)`
`DBL_MIN_EXP`	`(-1021)`
`LDBL_MIN_EXP` (AArch32)	`(-1021)`
`LDBL_MIN_EXP` (AArch64)	`16381`
`FLT_MIN_10_EXP`	`(-37)`
`DBL_MIN_10_EXP`	`(-307)`
`LDBL_MIN_10_EXP` (AArch32)	`(-307)`
`LDBL_MIN_10_EXP` (AArch64)	`4931`

Macro name	Value
`FLT_MAX_EXP`	`128`
`DBL_MAX_EXP`	`1024`
`LDBL_MAX_EXP` (AArch32)	`1024`
`LDBL_MAX_EXP` (AArch64)	`16384`
`FLT_MAX_10_EXP`	`38`
`DBL_MAX_10_EXP`	`308`
`LDBL_MAX_10_EXP` (AArch32)	`308`
`LDBL_MAX_10_EXP` (AArch64)	`4932`
`FLT_MAX`	`3.40282347e+38F`
`DBL_MAX`	`1.79769313486231571e+308`
`LDBL_MAX` (AArch32)	`1.79769313486231571e+308L`
`LDBL_MAX` (AArch64)	`1.18973149535723176508575932662800702e+4932L`
`FLT_EPSILON`	`1.19209290e-7F`
`DBL_EPSILON`	`2.2204460492503131e-16`
`LDBL_EPSILON` (AArch32)	`2.2204460492503131e-16L`
`LDBL_EPSILON` (AArch64)	`1.92592994438723585305597794258492732e-34L`
`FLT_MIN`	`1.175494351e-38F`
`DBL_MIN`	`2.22507385850720138e-308`
`LDBL_MIN` (AArch32)	`2.22507385850720138e-308L`
`LDBL_MIN` (AArch64)	`3.36210314311209350626267781732175260e-4932L`
`FLT_TRUE_MIN`	-
`DBL_TRUE_MIN`	-
`LDBL_TRUE_MIN` (AArch32)	-
`LDBL_TRUE_MIN` (AArch64)	-

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`

Macro name	Value
`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)`

Macro name	Value
`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`
`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`	`(9223372036854775807ll)`
`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` (AArch32 state)	4	4
`long` (AArch64 state)	8	8
`long long`	8	8
`float`	4	4
`double`	8	8
`long double` (AArch32 state)	8	8
`long double` (AArch64 state)	16	16

9. Standard C++ Implementation Definition

Provides information required by the ISO C++ Standard for conforming C++ implementations.

9.1 C++ Implementation definition

The ISO C++ Standard (ISO/IEC 14882:2014) defines the concept of implementation-defined behavior as the "behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents".

The following topics document the behavior in the implementation of Arm® Compiler 6 of the implementation-defined features of the C++ language. Each topic provides information from a single chapter in the C++ Standard. The C++ Standard section number relevant to each implementation-defined aspect is provided in parentheses.

9.2 General

Describes general implementation-defined aspects of the Arm C++ compiler and C++ library, as required by the ISO C++ Standard.

How a diagnostic is identified (1.3.6).

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 `-Wname`, only for messages that can be suppressed.

Libraries in a freestanding implementation (1.4).

Arm® Compiler supports the C99 and the C++11 standard libraries.

Bits in a byte (1.7).

The number of bits in a byte is 8.

What constitutes an interactive device (1.9).

What constitutes an interactive device depends on the environment and what the `_sys_istty` function reports. 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.

Related information

-W on page 89

9.3 Lexical conventions

Describes the lexical conventions of implementation-defined aspects of the Arm C++ compiler and C++ library, as required by the ISO C++ Standard.

Mapping of the physical source file characters to the basic source character set (2.2).

The input files are encoded in UTF-8. Due to the design of UTF-8 encoding, the basic source character set is represented in the source file in the same way as the ASCII encoding of the basic character set.

Physical source file characters (2.2).

The source file characters are encoded in UTF-8.

Conversion of characters from source character set to execution character set (2.2).

The source character set and the execution character set are the same.

Requirement of source for translation units when locating template definitions (2.2).

When locating the template definitions related to template instantiations, the source of the translation units that define the template definitions is not required.

Values of execution character sets (2.3).

Both the execution character set and the wide execution character set consist of all the code points defined by ISO/IEC 10646.

Mapping the header name to external source files (2.8).

In both forms of the `#include` preprocessing directive, the character sequences that specify header names are mapped to external header source file names.

Semantics of non-standard escape sequences (2.13.3).

The following non-standard escape sequences are accepted for compatibility with GCC:

Escape sequence	Code point
`\e`	`U+001B`
`\E`	`U+001B`

Value of wide-character literals containing multiple characters (2.13.3).

If a wide-character literal contains more than one character, only the right-most character in the literal is used.

Value of an ordinary character literal outside the range of its corresponding type (2.13.3).

This case is diagnosed and rejected.

Floating literals (2.13.4).

For a floating literal whose scaled value cannot be represented as a floating-point value, the nearest even floating-point value is chosen.

String literal concatenation (2.13.5).

Differently prefixed string literal tokens cannot be concatenated, except for the ones specified by the ISO C++ Standard.

9.4 Basic concepts

Describes basic concepts relating to implementation-defined aspects of the Arm C++ compiler and C++ library, as required by the ISO C++ Standard.

Start-up and termination in a freestanding environment (3.6.1).

The Arm® Compiler *Arm C and C++ Libraries and Floating-Point Support User Guide* describes the start-up and termination of programs.

Definition of main in a freestanding environment (3.6.1).

The `main` function must be defined.

Linkage of the main function (3.6.1).

The `main` function has external linkage.

Parameters of main (3.6.1).

The only permitted parameters for definitions of `main` of the form `int main(`*`parameters`*`)` are void and `int, char**`.

Dynamic initialization of static objects (3.6.2).

Static objects are initialized before the first statement of `main`.

Dynamic initialization of thread-local objects (3.6.2).

Thread-local objects are initialized at the first odr-use.

Pointer safety (3.7.4.3).

This implementation has relaxed pointer safety.

Extended signed integer types (3.9.1).

No extended integer types exist in the implementation.

Representation and signedness of the `char` type (3.9.1).

The `char` type is `unsigned` and has the same values as `unsigned char`.

Representation of the values of floating-point types (3.9.1).

The values of floating-point types are represented using the IEEE format as follows:

- `float` values are represented by IEEE single-precision values.
- `double` values are represented by IEEE double-precision values.
- `long double` values in AArch32 are represented by IEEE double-precision values.
- `long double` values in AArch64 are represented by IEEE quadruple-precision values.

Note

The `long double` data type is not supported for AArch64 state because of limitations in the current Arm C library.

Representation of values of pointer type (3.9.2).

Values of pointer type are represented as 32-bit addresses in AArch32 state and 64-bit addresses in AArch64 state.

Support of extended alignments (3.11).

Alignments, including extended alignments, that are a power of two and are less than or equal to `0x10000000` are supported.

Related information

Arm C and C++ Libraries and Floating-Point Support User Guide

9.5 Standard conversions

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to standard conversions, as required by the ISO C++ Standard.

Conversion to `signed` integer (4.7).

When an integer value is converted to a value of `signed` integer type, but cannot be represented by the destination type, the value is truncated to the number of bits of the destination type and then reinterpreted as a value of the destination type.

Result of inexact floating-point conversions (4.8).

When a floating-point value is converted to a value of a different floating-point type, and the value is within the range of the destination type but cannot be represented exactly, the value is rounded to the nearest floating-point value by default.

Result of inexact integer to floating-point conversion (4.9).

When an integer value is converted to a value of floating-point type, and the value is within the range of the destination type but cannot be represented exactly, the value is rounded to the nearest floating-point value by default.

9.6 Expressions

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to expressions, as required by the ISO C++ Standard.

Passing an argument of class type in a function call through ellipsis (5.2.2).

For ellipsis arguments, passing an argument of class type having a non-trivial copy constructor, a non-trivial move constructor, or a non-trivial destructor, with no corresponding parameter, results in an abort at run time. A diagnostic is reported for this case.

Result type of typeid expression (5.2.8).

The type of a typeid expression is an expression with dynamic type `std::type_info`.

Incrementing a bit-field that cannot represent the incremented value (5.2.6).

The incremented value is truncated to the number of bits in the bit-field. The bit-field is updated with the bits of the truncated value.

Conversions between pointers and integers (5.2.10).

Converting a pointer to an integer type with a smaller bit width than the pointer, truncates the pointer to the number of bits of the destination type. Converting a pointer to an integer type with a greater bit width than the pointer, zero-extends the pointer. Otherwise, the bits of the representation are unchanged.

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

Conversions from function pointers to object pointers (5.2.10).

Such conversions are supported.

`sizeof` applied to fundamental types other than `char`, `signed char`, and `unsigned char` (5.3.3).

Type	`sizeof`
`bool`	1
`char`	1
`wchar_t`	4
`char16_t`	2
`char32_t`	4
`short`	2
`int`	4
`long` (AArch32 state)	4
`long` (AArch64 state)	8
`long long`	8
`float`	4
`double`	8
`long double` (AArch32 state)	8
`long double` (AArch64 state)	16

Support for over-aligned types in `new` expressions (5.3.4).

Over-aligned types are not supported in `new` expressions. The pointer for the allocated type does not fulfill the extended alignment.

Type of `ptrdiff_t` (5.7).

The type of `ptrdiff_t` is signed int for AArch32 state and signed long for AArch64 state.

Type of `size_t` (5.7).

The type of `size_t` is unsigned int for AArch32 state and unsigned long for AArch64 state.

Result of right shift of negative value (5.8).

In a bitwise right shift operation of the form `E1 >> E2`, if `E1` is of signed type and has a negative value, the value of the result is the integral part of the quotient of `E1 / (2 ** E2)`, except when `E1` is -1, then the result is -1.

Assignment of a value to a bit-field that the bit-field cannot represent (5.18).

When assigning a value to a bit-field that the bit-field cannot represent, the value is truncated to the number of bits of the bit-field. A diagnostic is reported in some cases.

9.7 Declarations

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to declarations, as required by the ISO C++ Standard.

Meaning of attribute declaration (7).

Arm® Compiler 6 is based on LLVM and Clang technology. Clang defines several attributes as specified by the Clang documentation at https://clang.llvm.org/docs/AttributeReference.html.

From these attributes, Arm Compiler 6 supports attributes that are scoped with `gnu::` (for compatibility with GCC) and `clang::`.

Underlying type for enumeration (7.2).

The underlying type for enumerations without a fixed underlying type is `int` or `unsigned int`, depending on the values of the enumerators. The `-fshort-enums` command-line option uses the smallest unsigned integer possible, or the smallest signed integer possible if any enumerator is negative, starting with `char`.

Meaning of an `asm` declaration (7.4).

An `asm` declaration enables the direct use of T32, A32, or A64 instructions.

Semantics of linkage specifiers (7.5).

Only the string-literals "C" and "C++" can be used in a linkage specifier.

9.8 Declarators

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to declarators, as required by the ISO C++ Standard.

String resulting from `__func__` (8.4.1).

The value of `__func__` is the same as in C99.

Initialization of a bit-field with a value that the bit-field cannot represent (8.5).

When initializing a bit-field with a value that the bit-field cannot represent, the value is truncated to the number of bits of the bit-field. A diagnostic is reported in some cases.

Allocation of bit-fields within a class (9.6).

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.

Alignment of bit-fields within a class (9.6).

The storage unit containing the bit-fields is aligned to the alignment of the type of the bit-field.

9.9 Templates

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to templates, as required by the ISO C++ Standard.

Linkage specification in templates (14).

Only the linkage specifiers "C" and "C++" can be used in template declarations.

9.10 Exception handling

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to exception handling, as required by the ISO C++ Standard.

Stack unwinding before calling `std::terminate` when no suitable catch handler is found (15.3).

The stack is not unwound in this case.

Stack unwinding before calling `std::terminate` when a `noexcept` specification is violated (15.5.1).

The stack is unwound in this case.

9.11 C++ 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.

Numeric values of character literals in `#if` preprocessing directives (16.1).

Numeric values of character literals match the values that they have in expressions other than the `#if` or `#elif` preprocessing directives.

Sign of character literals in `#if` preprocessing directives (16.1).

Character literals in `#if` preprocessing directives are never negative.

Manner in which `#include <...>` source files are searched (16.2).

- If the character sequence begins with the `/` character, it is interpreted as an absolute file path.

- Otherwise, the character sequence is interpreted as a file path relative to one of the following directories:
  - The sequence of the directories specified using the `-I` command-line option, in the command-line order.
  - The `include` subdirectory in the compiler installation directory.

Manner in which `#include "..."` source files are searched (16.2).

- If the character sequence begins with the `/` character, it is interpreted as an absolute file path.
- Otherwise, the character sequence is interpreted as a file path relative to the parent directory of the source file that contains the `#include` preprocessing directive.

Nesting limit for `#include` preprocessing directives (16.2).

Limited only by the memory available at translation time.

Meaning of pragmas (16.6).

Arm® Compiler 6 is based on LLVM and Clang technology. Clang defines several pragmas as specified by the Clang documentation at http://clang.llvm.org/docs/LanguageExtensions.html.

Definition and meaning of `__STDC__` (16.8).

`__STDC__` is predefined as `#define __STDC__ 1`.

Definition and meaning of `__STDC_VERSION__` (16.8).

This macro is not predefined.

Text of `__DATE__` and `__TIME__` when the date or time of a translation is not available (16.8).

The date and time of the translation are always available on all supported platforms.

9.12 Library introduction

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the library introduction, as required by the ISO C++ Standard.

Linkage of names from the Standard C library (17.6.2.3).

Declarations from the C library have "C" linkage.

Library functions that can be recursively reentered (17.6.5.8).

Functions can be recursively reentered, unless specified otherwise by the ISO C++ Standard.

Exceptions thrown by C++ Standard Library functions that do not have an exception specification (17.6.5.12).

These functions do not throw any additional exceptions.

Errors category for errors originating from outside the operating system (17.6.5.14).

There is no additional error category.

9.13 Language support library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the language support library, as required by the ISO C++ Standard.

Exit status (18.5).

Control is returned to the host environment using the `_sys_exit` function of the Arm® C Library.

Returned value of `std::bad_alloc::what` (18.6.2.1).

The returned value is `std::bad_alloc`.

Returned value of `std::type_info::name` (18.7.1).

The returned value is a string containing the mangled name of the type that is used in the typeid expression. The name is mangled following the Itanium C++ ABI specification.

Returned value of `std::bad_cast::what` (18.7.2).

The returned value is `std::bad_cast`.

Returned value of `std::bad_typeid::what` (18.7.3).

The returned value is `std::bad_typeid`.

Returned value of `std::bad_exception::what` (18.8.1).

The returned value is `std::bad_exception`.

Returned value of `std::exception::what` (18.8.1).

The returned value is `std::exception`.

Use of non-POFs as signal handlers (18.10).

Non Plain Old Functions (POFs) can be used as signal handlers if no uncaught exceptions are thrown in the handler, and the execution of the signal handler does not trigger undefined behavior. For example, the signal handler may have to call `std::_Exit` instead of `std::exit`.

9.14 General utilities library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the general utilities library, as required by the ISO C++ Standard.

Return value of `std::get_pointer_safety` (20.7.4).

This function always returns `std::pointer_safety::relaxed`.

Support for over-aligned types by the allocator (20.7.9.1).

The allocator does not support over-aligned types.

Support for over-aligned types by `get_temporary_buffer` (20.7.11).

Function `std::get_temporary_buffer` does not support over-aligned types.

Returned value of `std::bad_weak_ptr::what` (20.8.2.2.1).

The returned value is `bad_weak_ptr`.

Exception type when the constructor of `std::shared_ptr` fails (20.8.2.2.1).

`std::bad_alloc` is the only exception that the `std::shared_ptr` constructor throws that receives a pointer.

Placeholder types (20.9.10.4).

Placeholder types, such as `std::placeholders::_1`, are not `CopyAssignable`.

Over-aligned types and type traits `std::aligned_storage` and `std::aligned_union` (20.10.7.6).

These two traits support over-aligned types.

Conversion between `time_t` and `time_point` (20.12.7.1).

The values are truncated in either case.

9.15 Strings library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the strings library, as required by the ISO C++ Standard.

Type of `std::streamoff` (21.2.3.1).

Type `std::streamoff` has type `long long`.

Type of `std::streampos` (21.2.3.2).

Type of `std::streampos` is `fpos<mbstate_t>`.

Returned value of `char_traits<char16_t>::eof` (21.2.3.2).

This function returns `uint_least16_t(0xFFFF)`.

Type of `std::u16streampos` (21.2.3.3).

Type of `std::u16streampos` is `fpos<mbstate_t>`.

Returned value of `char_traits<char32_t>::eof` (21.2.3.3).

This function returns `uint_least32_t(0xFFFFFFFF)`.

Type of `std::u32streampos` (21.2.3.3).

Type of `std::u32streampos` is `fpos<mbstate_t>`.

Type of `std::wstreampos` (21.2.3.4).

Type of `std::wstreampos` is `fpos<mbstate_t>`.

Supported multibyte character encoding rules (21.2.3.4).

UTF-8 and Shift-JIS are supported as multibyte character encodings.

9.16 Localization library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the localization library, as required by the ISO C++ Standard.

Locale object (22.3.1.2).

There is one global locale object for the entire program.

Permitted locale names (22.3.1.2).

Valid locale values depend on which `__use_X_ctype` symbols are imported (`__use_iso8859_ctype`, `__use_sjis_ctype`, `__use_utf8_ctypte`), and on user-defined locales.

Effect on C locale of calling `locale::global` (22.3.1.5).

Calling this function with an unnamed locale has no effect.

Value of `ctype<char>::table_size` (22.4.1.3.1).

The value of `ctype<char>::table_size` is 256.

Two-digit year numbers in the function `std::time_get::do_get_year` (22.4.5.1.2).

Two-digit year numbers are accepted. Years from 00 to 68 are assumed to mean years 2000 to 2068, while years from 69 to 99 are assumed to mean 1969 to 1999.

Additional formats for `std::time_get::do_get_date` (22.4.5.1.2).

No additional formats are defined.

Formatted character sequence that `std::time_put::do_put` generates in the C locale (22.4.5.3.2).

The behavior is the same as that of the Arm C library function `strftime`.

Mapping from name to catalog when calling `std::messages::do_open` (22.4.7.1.2).

No mapping happens as this function does not open any catalog.

Mapping to message when calling `std::messages::do_get` (22.4.7.1.2).

No mapping happens and `dflt` is always returned.

9.17 Containers library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the containers library, as required by the ISO C++ Standard.

Type of `std::array::iterator` and `std::array::const_iterator` (23.3.2.1).

The types of `std::array<T>::iterator` and `std::array<T>::const_iterator` are `T*` and `const T*` respectively.

Default number of buckets in `std::unordered_map` (23.5.4.2).

When constructing a container with an iterator range and without specifying the number of buckets, the number of buckets that are used is equal to the size of the iterator range. Every element of the iterator range is inserted in an empty container.

Default number of buckets in `std::unordered_multimap` (23.5.4.2).

When constructing a container with an iterator range and without specifying the number of buckets, the number of buckets that are used is equal to the size of the iterator range. Every element of the iterator range is inserted in an empty container.

Default number of buckets in `std::unordered_set` (23.5.6.2).

When constructing a container with an iterator range and without specifying the number of buckets, the number of buckets that are used is equal to the size of the iterator range. Every element of the iterator range is inserted in an empty container.

Default number of buckets in `std::unordered_multiset` (23.5.7.2).

When constructing a container with an iterator range and without specifying the number of buckets, the number of buckets that are used is equal to the size of the iterator range. Every element of the iterator range is inserted in an empty container.

9.18 Input/output library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the input/output library, as required by the ISO C++ Standard.

Behavior of iostream classes when `traits::pos_type` is not streampos or when `traits::off_type` is not streamoff (27.2.1).

There is no specific behavior implemented for this case.

Effect of calling `std::ios_base::sync_with_stdio` after any input or output operation on standard streams (27.5.3.4).

Previous input/output is not handled in any special way.

Exception thrown by `basic_ios::clear` (27.5.5.4).

When `basic_ios::clear` throws as exception, it throws an exception of type `basic_ios::failure` constructed with `"ios_base::clear"`.

Move constructor of `std::basic_stringbuf` (27.8.2.1).

The constructor copies the sequence pointers.

Effect of calling `std::basic_filebuf::setbuf` with nonzero arguments (27.9.1.2).

The provided buffer replaces the internal buffer. The object can use up to the provided number of bytes of the buffer.

Effect of calling `std::basic_filebuf::sync` when a get area exists (27.9.1.5).

The get area is emptied and the current file position is moved back the corresponding number of bytes.

9.19 Regular expressions library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the regular expressions library, as required by the ISO C++ Standard.

Type of `std::regex_constants::error_type`

The enum `std::regex_constants::error_type` is defined as follows:

```
enum error_type
{
    error_collate = 1,
    error_ctype,
    error_escape,
    error_backref,
    error_brack,
    error_paren,
    error_brace,
    error_badbrace,
    error_range,
    error_space,
    error_badrepeat,
    error_complexity,
    error_stack,
    __re_err_grammar,
    __re_err_empty,
    __re_err_unknown
};
```

9.20 Atomic operations library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the atomic operations library, as required by the ISO C++ Standard. These macros are defined in the `include/libcxx/atomic` header file.

Values of `ATOMIC_...LOCK_FREE` macros (29.4)

Macro	Value
ATOMIC_BOOL_LOCK_FREE	2
ATOMIC_CHAR_LOCK_FREE	2
ATOMIC_CHAR16_T_LOCK_FREE	2
ATOMIC_CHAR32_T_LOCK_FREE	2
ATOMIC_WCHAR_T_LOCK_FREE	2
ATOMIC_SHORT_LOCK_FREE	2
ATOMIC_INT_LOCK_FREE	2
ATOMIC_LONG_LOCK_FREE	2
ATOMIC_LLONG_LOCK_FREE	2
ATOMIC_POINTER_LOCK_FREE	2

9.21 Thread support library

Describes implementation-defined aspects of the Arm C++ compiler and C++ library relating to the thread support library, as required by the ISO C++ Standard.

Presence and meaning of `native_handle_type` and `native_handle`.

The library uses the following native handles as part of the thread portability mechanism, which is described elsewhere.

```
__ARM_TPL_mutex_t used in std::mutex and std::recursive_mutex

__ARM_TPL_condvar_t used in std::condition_variable

__ARM_TPL_thread_id used in std::thread

__ARM_TPL_thread_t used in std::thread
```

9.22 Implementation quantities

Describes limits in C++ implementations.

Note

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

Note

Where a specific number is provided, this value is the recommended minimum quantity.

Nesting levels of compound statements, iteration control structures, and selection control structures.
256. Can be increased using the `-fbracket-depth` command-line option.

Nesting levels of conditional inclusion
Limited by memory.

Pointer, array, and function declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration.
Limited by memory.

Nesting levels of parenthesized expressions within a full-expression.
256. Can be increased using the `-fbracket-depth` command-line option.

Number of characters in an internal identifier or macro name.
Limited by memory.

Number of characters in an external identifier.
Limited by memory.

External identifiers in one translation unit.
Limited by memory.

Identifiers with block scope declared in one block.
Limited by memory.

Macro identifiers that are simultaneously defined in one translation unit.
Limited by memory.

Parameters in one function definition.
Limited by memory.

Arguments in one function call.
Limited by memory.

Parameters in one macro definition.
Limited by memory.

Arguments in one macro invocation.
Limited by memory.

Characters in one logical source line.
Limited by memory.

Characters in a string literal (after concatenation).
Limited by memory.

Size of an object.
`SIZE_MAX`

Nesting levels for #include files.
Limited by memory.

Case labels for a switch statement (excluding case labels for any nested switch statements).
Limited by memory.

Data members in a single class.
Limited by memory.

Enumeration constants in a single enumeration.
Limited by memory.

Levels of nested class definitions in a single member-specification.
256. Can be increased using the `-fbracket-depth` command-line option.

Functions that are registered by `atexit()`.
Limited by memory.

Direct and indirect base classes.
Limited by memory.

Direct base classes for a single class.
Limited by memory.

Members declared in a single class.
Limited by memory.

Final overriding virtual functions in a class, accessible or not.
Limited by memory.

Direct and indirect virtual bases of a class.
Limited by memory.

Static members of a class.
Limited by memory.

Friend declarations in a class.
Limited by memory.

Access control declarations in a class.
Limited by memory.

Member initializers in a constructor definition.
Limited by memory.

Scope qualifications of one identifier.
Limited by memory.

Nested external specifications.
Limited by memory.

Recursive `constexpr` function invocations.
512. Can be changed using the [COMMUNITY] command-line option, `-fconstexpr-depth`.

Full-expressions that are evaluated within a core constant expression.
Limited by memory.

Template arguments in a template declaration.
Limited by memory.

Recursively nested template instantiations, including substitution during template argument deduction (14.8.2).
1024. Can be changed using the [COMMUNITY] command-line option, `-ftemplate-depth`.

Handlers per try block.
Limited by memory.

Throw specifications on a single function declaration.
Limited by memory.

Number of placeholders (20.9.10.4).
Ten placeholders from `_1` to `_10`.

10. armclang Integrated Assembler

Provides information on integrated assembler features, such as the directives you can use when writing assembly language source files in the armclang integrated assembler syntax.

10.1 Syntax of assembly files for integrated assembler

Assembly statements can include labels, instructions, directives, or macros.

Syntax

```
label:
    instruction[;]
    directive[;]
    macro_invocation[;]
```

Description

label

For label statements, the statement ends after the `:` character. For the other forms of assembler statements, the statement ends at the first newline or `;` character. This means that any number of labels can be defined on the same source line, and multiple of any other types of statements can be present in one source line if separated by `;`.

Label names without double quotes:

- Must start with a period (`.`), `_`, `a-z` or `A-Z`.
- Can also contain numbers, `_`, `$`.
- Must not contain white spaces.

You can have white spaces in label names by surrounding them with double quotes. Escape sequences are not interpreted within label names. It is also not possible to have double quotes as part of the label name.

instruction

The optional `;` can be used to end the statement and start a new statement on the same line.

directive

The optional `;` can be used to end the statement and start a new statement on the same line.

macro_invocation

The optional `;` can be used to end the statement and start a new statement on the same line.

Comments

Comments are treated as equivalent to whitespace, their contents are ignored by the assembler.

There are two ways to include comments in an assembly file:

```
// single-line comment
@ single-line comment in AArch32 state only
/* multi-line
   comment */
```

In single-line comments, the `//` marker starts a comment that runs to the end of the source line. Unlike when compiling C and C++ source, the end of the line cannot be escaped with `\` to continue the comment.

`@` starts a single-line comment in AArch32 state. `@` is not a comment character in AArch64 state.

In multi-line comments, the `/*` marker starts a comment that runs to the first occurrence of `*/`, even if that is on a later line. Like in C and C++ source, the comment always ends at the first `*/`, so comments cannot be nested. This style of comments can be used anywhere within an assembly statement where whitespace is valid.

Examples

```
      // Instruction on it's own line:
      add r0, r1, r2

     // Label and directive:
lab:  .word 42

     // Multiple labels on one line:
lab1: lab2:

      /* Multiple instructions, directives or macro-invocations
         must be separated by ';' */
      add r0, r1, r2; bx lr

      // Multi-line comments can be used anywhere whitespace can:
      add /*dst*/r0, /*lhs*/r1, /*rhs*/r2
```

10.2 Assembly expressions

Expressions consist of one or more integer literals or symbol references, combined using operators.

You can use an expression when an instruction operand or directive argument expects an integer value or label.

Not all instruction operands and directive arguments accept all possible expressions. For example, the alignment directives require an absolute expression for the boundary to align to. Therefore, alignment directives cannot accept expressions involving labels, but can accept expressions involving only integer constants.

On the other hand, the data definition directives can accept a wider range of expressions, including references to defined or undefined symbols. However, the types of expressions accepted is still limited by the ELF relocations available to describe expressions involving undefined symbols. For example, it is not possible to describe the difference between two symbols defined in different sections. The assembler reports an error when an expression is not valid in the context in which it is used.

Expressions involving integer constants are evaluated as signed 64-bit values internally to the assembler. If an intermediate value in a calculation cannot be represented in 64 bits, the behavior is undefined. The assembler does not currently emit a diagnostic when this happens.

Constants

Numeric literals are accepted in the following formats:

- Decimal integer in range `0` to (2^{64})-1.
- Hexadecimal integer in range `0` to (2^{64})-1, prefixed with `0x`.
- Octal integer in range `0` to (2^{64})-1, prefixed with `0`.
- Binary integer in range `0` to (2^{64})-1, prefixed with `0b`.

Some directives accept values larger than (2^{64})-1. These directives only accept simple integer literals, not expressions.

Note

These ranges do not include negative numbers. Negative numbers can instead be represented using the unary operator, `-`.

Symbol References

References to symbols are accepted as expressions. Symbols do not need to be defined in the same assembly language source file, to be referenced in expressions.

The period symbol (`.`) is a special symbol that can be used to reference the current location in the output file.

For AArch32 targets, a symbol reference might optionally be followed by a modifier in parentheses. The following modifiers are supported:

Table 10-1: Modifiers

Modifier	Meaning
None	Do not relocate this value.
`got_prel`	Offset from this location to the GOT entry of the symbol.
`target1`	Defined by platform ABI.
`target2`	Defined by platform ABI.
`prel31`	Offset from this location to the symbol. Bit 31 is not modified.
`sbrel`	Offset to symbol from addressing origin of its output segment.
`got`	Address of the GOT entry for the symbol.
`gotoff`	Offset from the base of the GOT to the symbol.

Operators

The following operators are valid expressions:

Table 10-2: Unary operators

Unary operator	Meaning
`-expr`	Arithmetic negation of `expr`.
`+expr`	Arithmetic addition of `expr`.
`~expr`	Bitwise negation of `expr`.

Table 10-3: Binary operators

Binary operator	Meaning
`expr1 - expr2`	Subtraction.
`expr1 + expr2`	Addition.
`expr1 * expr2`	Multiplication.
`expr1 / expr2`	Division.
`expr1 % expr2`	Modulo.

Table 10-4: Binary logical operators

Binary logical operator	Meaning
`expr1 && expr2`	Logical and. 1 if both operands non-zero, 0 otherwise.
`expr1 || expr2`	Logical or. 1 if either operand is non-zero, 0 otherwise.

Table 10-5: Binary bitwise operators

Binary bitwise operator	Meaning
`expr1 & expr2`	`expr1` bitwise and `expr2`.
`expr1 | expr2`	`expr1` bitwise or `expr2`.
`expr1 ^ expr2`	`expr1` bitwise exclusive-or `expr2`.
`expr1 >> expr2`	Logical shift right `expr1` by `expr2` bits.
`expr1 << expr2`	Logical shift left `expr1` by `expr2` bits.

Table 10-6: Binary comparison operators

Binary comparison operator	Meaning
`expr1 == expr2`	`expr1` equal to `expr2`.
`expr1 != expr2`	`expr1` not equal to `expr2`.
`expr1 < expr2`	`expr1` less than `expr2`.
`expr1 > expr2`	`expr1` greater than `expr2`.
`expr1 <= expr2`	`expr1` less than or equal to `expr2`.
`expr1 >= expr2`	`expr1` greater than or equal to `expr2`.

The order of precedence for binary operators is as follows, with highest precedence operators listed first:

1. `*`, `/`, `%`, `>>`, `<<`

2. `|`, `^`, `&`
3. `+`, `-`
4. `==`, `!=`, `<`, `>`, `<=`, `>=`
5. `&&`
6. `||`

Operators listed on the same line have equal precedence, and are evaluated from left to right. All unary operators have higher precedence than any binary operators.

Note

The precedence rules for assembler expressions are not identical to those for C.

Relocation specifiers

For some instruction operands, a relocation specifier might be used to specify which bits of the expression must be used for the operand, and which type of relocation must be used.

These relocation specifiers can only be used at the start of an expression. They can only be used in operands of instructions that support them.

In AArch32 state, the following relocation specifiers are available:

Table 10-7: Relocation specifiers for AArch32 state

Relocation specifier	Meaning
`:lower16:`	Use the lower 16 bits of the expression value.
`:upper16:`	Use the upper 16 bits of the expression value.

These relocation specifiers are only valid for the operands of the `movw` and `movt` instructions. They can be combined with an expression involving the current place to create a place-relative relocation, and with the `sbrel` symbol modifier to create a static-base-relative relocation. The current place is the location that the assembler is emitting code or data at. A place-relative relocation is a relocation that generates the offset from the relocated data to the symbol it references.

In AArch64 state, the following relocation specifiers are available:

Table 10-8: Relocation specifiers for AArch64 state

Relocation specifier	Relocation type	Bits to use	Overflow checked
`:lo12:`	Absolute	[11:0]	No
`:abs_g3:`	Absolute	[63:48]	Yes
`:abs_g2:`	Absolute	[47:32]	Yes
`:abs_g2_s:`	Absolute, signed	[47:32]	Yes
`:abs_g2_nc:`	Absolute	[47:32]	No

Relocation specifier	Relocation type	Bits to use	Overflow checked
`:abs_g1:`	Absolute	[31:16]	Yes
`:abs_g1_s:`	Absolute, signed	[31:16]	Yes
`:abs_g1_nc:`	Absolute	[31:16]	No
`:abs_g0:`	Absolute	[15:0]	Yes
`:abs_g0_s:`	Absolute, signed	[15:0]	Yes
`:abs_g0_nc:`	Absolute	[15:0]	No
`:got:`	Global Offset Table Entry	[32:12]	Yes
`:got_lo12:`	Global Offset Table Entry	[11:0]	No

These relocation specifiers can only be used in the operands of instructions that have matching relocations defined in ELF for the Arm 64-bit Architecture (AArch64). They can be combined with an expression involving the current place to create a place-relative relocation.

Examples

```
      // Using an absolute expression in an instruction operand:
      orr r0, r0, #1<<23

      // Using an expression in the memory operand of an LDR instruction to
      // reference an offset from a symbol.
func:
      ldr r0, #data+4 // Loads 2 into r0
      bx lr
data:
      .word 1
      .word 2

      // Creating initialized data that contains the distance between two
      // labels:
size:
      .word end - start
start:
      .word 123
      .word 42
      .word 4523534
end:

      // Load the base-relative address of 'sym' (used for 'RWPI'
      // position-independent code) into r0 using movw and movt:
      movw r0, #:lower16:sym(sbrel)
      movt r0, #:upper16:sym(sbrel)

      // Load the address of 'sym' from the GOT using ADRP and LDR (used for
      // position-independent code on AArch64):
      adrp x0, #:got:sym
      ldr x0, [x0, #:got_lo12:sym]

      // Constant pool entry containing the offset between the location and a
      // symbol defined elsewhere. The address of the symbol can be calculated
      // at runtime by adding the value stored in the location of the address
      // of the location. This is one technique for writing position-
      // independent code, which can be executed from an address chosen at
      // runtime without re-linking it.
      adr r0, address
      ldr r1, [r0]
      add r0, r0, r1
address:
      .word extern_symbol - .
```

10.3 Alignment directives

The alignment directives align the current location in the file to a specified boundary.

Syntax

```
.balign num_bytes [, fill_value]

.balignl num_bytes [, fill_value]

.balignw num_bytes [, fill_value]

.p2align exponent [, fill_value]

.p2alignl exponent [, fill_value]

.p2alignw exponent [, fill_value]

.align exponent [, fill_value]
```

Description

num_bytes

This specifies the number of bytes that must be aligned to. This must be a power of 2.

exponent

This specifies the alignment boundary as an exponent. The actual alignment boundary is 2^{exponent}.

fill_value

The value to fill any inserted padding bytes with. This value is optional.

Operation

The alignment directives align the current location in the file to a specified boundary. The unused space between the previous and the new current location are filled with:

- Copies of *fill_value*, if it is specified. The width of *fill_value* can be controlled with the `w` and `l` suffixes, see below.
- NOP instructions appropriate to the current instruction set, if all the following conditions are specified:
  - The *fill_value* argument is not specified.
  - The `w` or `l` suffix is not specified.
  - The alignment directive follows an instruction.
- Zeroes otherwise.

The `.balign` directive takes an absolute number of bytes as its first argument, and the `.p2align` directive takes a power of 2. For example, the following directives align the current location to the next multiple of 16 bytes:

- `.balign 16`

- `.p2align 4`
- `.align 4`

The `w` and `l` suffixes modify the width of the padding value that is inserted.

- By default, the *`fill_value`* is a 1-byte value.
- If the `w` suffix is specified, the *`fill_value`* is a 2-byte value.
- If the `l` suffix is specified, the *`fill_value`* is a 4-byte value.

If either of these suffixes are specified, the padding values are emitted as data (defaulting to a value of zero), even if following an instruction.

The `.align` directive is an alias for `.p2align`, but it does not accept the `w` and `l` suffixes.

Alignment is relative to the start of the section in which the directive occurs. If the current alignment of the section is lower than the alignment requested by the directive, the alignment of the section is increased.

Usage

Use the alignment directives to ensure that your data and code are aligned to appropriate boundaries. This is typically required in the following circumstances:

- In T32 code, the ADR instruction and the PC-relative version of the LDR instruction can only reference addresses that are 4-byte aligned, but a label within T32 code might only be 2-byte aligned. Use `.balign 4` to ensure 4-byte alignment of an address within T32 code.
- Use alignment directives to take advantage of caches on some Arm® processors. For example, many processors have an instruction cache with 16-byte lines. Use `.p2align 4` or `.balign 16` to align function entry points on 16-byte boundaries to maximize the efficiency of the cache.

Examples

Aligning a constant pool value to a 4-byte boundary in T32 code:

```
get_val:
      ldr r0, value
      adds r0, #1
      bx lr
      // The above code is 6 bytes in size.
      // Therefore the data defined by the .word directive below must be manually
 aligned
      // to a 4-byte boundary to be able to use the LDR instruction.
      .p2align 2
value:
      .word 42
```

Ensuring that the entry points to functions are on 16-byte boundaries, to better utilize caches:

```
      .p2align 4
      .type func1, "function"
func1:
      // code

      .p2align 4
```

```
        .type func2, "function"
func2:
        // code
```

Note

In both of the examples above, it is important that the directive comes before the label that is to be aligned. If the label came first, then it would point at the padding bytes, and not the function or data it is intended to point to.

10.4 Data definition directives

These directives allocate memory in the current section, and define the initial contents of that memory.

Syntax

```
    .byte expr[, expr]...

    .hword expr[, expr]...

    .word expr[, expr]...

    .quad expr[, expr]...

    .octa expr[, expr]...
```

Description

expr

An expression that has one of the following forms:

- A absolute value, or expression (not involving labels) which evaluates to one. For example:

```
    .word (1<<17) | (1<<6)
    .word 42
```

- An expression involving one label, which may or not be defined in the current file, plus an optional constant offset. For example:

```
    .word label
    .word label + 0x18
```

- A place-relative expression, involving the current location in the file (or a label in the current section) subtracted from a label which may either be defined in another section in the file, or undefined in the file. For example:

```
foo:
    .word label - .
    .word label - foo
```

- A difference between two labels, both of which are defined in the same section in the file. The section containing the labels need not be the same as the one containing the directive. For example:

```
    .word end - start
start:
        // ...
end:
```

The number of bytes allocated by each directive is as follows:

Table 10-9: Data definition directives

Directive	Size in bytes
`.byte`	1
`.hword`	2
`.word`	4
`.quad`	8
`.octa`	16

If multiple arguments are specified, multiple memory locations of the specified size are allocated and initialized to the provided values in order.

The following table shows which expression types are accepted for each directive. In some cases, this varies between AArch32 and AArch64. This is because the two architectures have different relocation codes available to describe expressions involving symbols defined elsewhere. For absolute expressions, the table gives the range of values that are accepted (inclusive on both ends).

Table 10-10: Expression types supported by the data definition directives

Directive	Absolute	Label	Place-relative	Difference
`.byte`	Within the range `[-128,255]` only	AArch32 only	Not supported	AArch64 and AArch32
`.hword`	Within the range `[-0x8000,0xffff]` only	AArch64 and AArch32	AArch64 only	AArch64 and AArch32
`.word`	Within the range `[-2^31,2^32-1]` only	AArch64 and AArch32	AArch64 and AArch32	AArch64 and AArch32
`.quad`	Within the range `[-2^63,2^64-1]` only	AArch64 only	AArch64 only	AArch64 only
`.octa`	Within the range `[0,2^128-1]` only	Not supported	Not supported	Not supported

Note

While most directives accept expressions, the `.octa` directive only accepts literal values. In the `armclang` inline assembler and integrated assembler, negative values are expressions (the unary negation operator and a positive integer literal), so negative values are not accepted by the `.octa` directive. If negative 16-byte values are needed, you can rewrite them using two's complement representation instead.

These directives do not align the start of the memory allocated. If this is required you must use one of the alignment directives.

The following aliases for these directives are also accepted:

Table 10-11: Aliases for the data definition directives

Directive	Aliases
`.byte`	`.1byte`, `.dc.b`
`.hword`	`.2byte`, `.dc`, `.dc.w`, `.short`, `.value`
`.word`	`.4byte`, `.long`, `.int`, `.dc.l`, `.dc.a` (AArch32 only)
`.quad`	`.8byte`, `.xword` (AArch64 only), `.dc.a` (AArch64 only)

Examples

```
    // 8-bit memory location, initialized to 42:
    .byte 42

    // 32-bit memory location, initialized to 15532:
    .word 15532

    // 32-bit memory location, initailized to the address of an externally defined
symbol:
    .word extern_symbol

    // 16-bit memory location, initialized to the difference between the 'start' and
    // 'end' labels. They must both be defined in this assembly file, and must be
    // in the same section as each other, but not necessarily the same section as
    // this directive:
    .hword end - start

    // 32-bit memory location, containing the offset between the current location in
the file and an externally defined symbol.
    .word extern_symbol - .
```

10.5 String definition directives

Allocates one or more bytes of memory in the current section, and defines the initial contents of the memory from a string literal.

Syntax

```
    .ascii "string"

    .asciz "string"

    .string "string"
```

Description

`.ascii`

The `.ascii` directive does not append a null byte to the end of the string.

`.asciz`

The `.asciz` directive appends a null byte to the end of the string.

The `.string` directive is an alias for `.asciz`.

`string`

The following escape characters are accepted in the string literal:

Table 10-12: Escape characters for the string definition directives

Escape character	Meaning
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\&`	Quote (`&`)
`\\`	Backslash (`\`)
`{Octal_Escape_Code}`	Three digit octal escape code for each ASCII character

Examples

Using a null-terminated string in a constant pool:

```
      .text
hello:
      adr r0, str_hello
      b printf
str_hello:
      .asciz "Hello, world!\n"
```

Generating pascal-style strings (which are prefixed by a length byte, and have no null terminator), using a macro to avoid repeated code. See also Macro directives and temporary numeric labels.

```
      .macro pascal_string, str
      .byte 2f - 1f
1:
      .ascii "\str"
2:
      .endm

      .data
hello:
      pascal_string "Hello"
goodbye:
      pascal_string "Goodbye"
```

10.6 Floating-point data definition directives

These directives allocate memory in the current section of the file, and define the initial contents of that memory using a floating-point value.

Syntax

```
      .float value [, value]...
```

```
    .double value [, value]...
```

Description

`.float`

The `.float` directive allocates 4 bytes of memory per argument, and stores the values in IEEE754 single-precision format.

`.double`

The `.double` directive allocates 8 bytes of memory per argument, and stores the values in IEEE754 double-precision format.

`value`

`value` is a floating-point literal.

Operation

If a floating-point value cannot be exactly represented by the storage format, it is rounded to the nearest representable value using the `round to nearest, ties to even` rounding mode.

The following aliases for these directives are also accepted:

Table 10-13: Aliases for the floating-point data definition directives

Directive	Alias
`.float`	`.single`, `.dc.s`
`.double`	`.dc.d`

Examples

```
float_pi:
    .float 3.14159265359
double_pi:
    .double 3.14159265359
```

10.7 Section directives

The section directives instruct the assembler to change the ELF section that code and data are emitted into.

Syntax

`.section` *`name`* `[, "`*`flags`*`" [, %`*`type`* `[,` *`entry_size`*`] [,` *`group_name`* `[,` *`linkage`*`]] [,` *`link_order_symbol`*`] [, ` *`unique`*`,` *`unique_id`*`] ]]`

`.pushsection .section` *`name`* `[, "`*`flags`*`" [, %`*`type`* `[,` *`entry_size`*`] [,` *`group_name`* `[,` *`linkage`*`]] [,` *`link_order_symbol`*`] [, ` *`unique`*`,` *`unique_id`*`] ]]`

`.popsection`

```
.text

.data

.rodata

.bss
```

Description

name

The *name* argument gives the name of the section to switch to.

By default, if the name is identical to a previous section, or one of the built-in sections, the assembler switches back to that section. Any code or data that is assembled is appended to the end of that section. The unique-id argument can be used to override this behavior.

flags

The optional *flags* argument is a quoted string containing any of the following characters, which correspond to the sh_flags field in the ELF section header.

Table 10-14: Section flags

Flag	Meaning
a	SHF_ALLOC: the section is allocatable.
w	SHF_WRITE: the section is writable.
y	SHF_ARM_PURECODE: the section is not readable.
x	SHF_EXECINSTR: the section is executable.
o	SHF_LINK_ORDER: the section has a link-order restriction.
M	SHF_MERGE: the section can be merged.
S	SHF_STRINGS: the section contains null-terminated string.
T	SHF_TLS: the section is thread-local storage.
G	SHF_GROUP: the section is a member of a section group.
?	if the previous section was part of a group, this section is in the same group, otherwise ignored.

The flags can be specified as a numeric value, with the same encoding as the sh_flags field in the ELF section header. This cannot be combined with the flag characters listed above. When using this syntax, the quotes around the flags value are still required.

Note

Certain flags need extra arguments, as described in the respective arguments.

`type`

The optional *`type`* argument is accepted with two different syntaxes: `%`*`type`* and `"`*`type`*`"`. It corresponds to the sh_type field in the ELF section header. The following values for the type argument are accepted:

Table 10-15: Section Type

Argument	ELF type	Meaning
`%progbits`	`SHT_PROGBITS`	Section contains initialized data and/or instructions.
`%nobits`	`SHT_NOBITS`	Section consists only of zero-initialized data.
`%note`	`SHT_NOTE`	Section contains information that the linker or loader use to check compatibility.
`%init_array`	`SHT_INIT_ARRAY`	Section contains an array of pointers to initialization functions.
`%fini_array`	`SHT_FINI_ARRAY`	Section contains an array of pointers to termination functions.
`%preinit_array`	`SHT_PREINIT_ARRAY`	Section contains an array of pointers to pre-initialization functions.

The type can be specified as a numeric value, with the same encoding as the sh_type field in the ELF section header. When using this syntax, the quotes around the type value are still required.

`entry_size`

If the `M` flag is specified, the *`entry_size`* argument is required. This argument must be an integer value, which is the size of the records that are contained within this section, that the linker can merge.

`group_name`

If the `G` flag is specified, the *`group_name`* argument is required. This argument is a symbol name to be used as the signature to identify the section group. All sections in the same object file and with the same *`group_name`* are part of the same section group.

If the `?` flag is specified, the section is implicitly in the same group as the previous section, and the *`group_name`* and *`linkage`* options are not accepted.

It is an error to specify both the G and ? flags on the same section.

`linkage`

If the `G` flag is specified, the optional linkage argument is allowed. The only valid value for this argument is `comdat`, which has the same effect as not providing the linkage argument. If any arguments after the group_name and linkage arguments are to be provided, then the linkage argument must be provided.

If the ? flag is specified, the section is implicitly in the same group as the previous section, and the group_name and linkage options are not accepted.

It is an error to specify both the G and ? flags on the same section.

`link_order_symbol`

If the `o` flag is specified, the *`link_order_symbol`* argument is required. This argument must be a symbol which is defined earlier in the same file. If multiple sections with the `o` flag are present at link time, the linker ensures that they are in the same order in the image as the sections that define the symbols they reference.

`unique` and `unique_id`

If the optional *`unique`* argument is provided, then the *`unique_id`* argument must also be provided. This argument must be a constant expression which evaluates to a positive integer. If a section has previously been created with the same name and unique ID, then the assembler switches to the existing section, appending content to it. Otherwise, a new section is created. Sections without a unique ID specified is never merged with sections that do have one. This allows creating multiple sections with the same name. The exact value of the unique ID is not important, and it has no effect on the generated object file.

Operation

The `.section` directive switches the current target section to the one described by its arguments. The `.pushsection` directive pushes the current target section onto a stack, and switches to the section described by its arguments. The `.popsection` directive takes no arguments, and reverts the current target section to the previous one on the stack. The rest of the directives (`.text`, `.data`, `.rodata`, `.bss`) switch to one of the built-in sections.

If continuing a previous section, and the flags, type, or other arguments do not match the previous definition of the section, then the arguments of the current `.section` directive has no effect on the section. Instead, the assembler uses the arguments from the previous `.section` directive. The assembler does not currently emit a diagnostic when this happens.

Default

Some section names and section name prefixes implicitly have some flags set. Extra flags can be set using the *`flags`* argument, but it is not possible to clear these implicit flags. The section names that have implicit flags are listed in the table here. For sections names not mentioned in the table, the default is to have no flags.

If the `%type` argument is not provided, the type is inferred from the section name. For sections names not mentioned in the table here, the default section type is `%progbits`.

Table 10-16: Sections with implicit flags and default types

Section name	Implicit Flags	Default Type
`.rodata`	a	`%progbits`
`.text`	ax	`%progbits`
`.init`	ax	`%progbits`
`.fini`	ax	`%progbits`
`.data`	aw	`%progbits`
`.bss`	aw	`%nobits`
`.init_array`	No default	`%init_array`
`.fini_array`	No default	`%fini_array`

Section name	Implicit Flags	Default Type
`.preinit_array`	No default	`%preinit_array`
`.tdata`	awT	`%progbits`
`.tbss`	awT	`%nobits`
`.note*`	No default	`%note`

Examples

- Splitting code and data into the built-in `.text` and `.data` sections. The linker can place these sections independently, for example to place the code in flash memory, and the writable data in RAM.

```
     .text
get_value:
      movw r0, #:lower16:value
      movt r0, #:upper16:value
      ldr r0, [r0]
      bx lr

      .data
value:
      .word 42
```

- Creating a section containing constant, mergeable records. This section contains a series of 8-byte records, where the linker is allowed to merge two records with identical content (possibly coming from different object files) into one record to reduce the image size.

```
      .section mergable, "aM", %progbits, 8
entry1:
      .word label1
      .word 42
entry2:
      .word label2
      .word 0x1234
```

- Creating two sections with the same name:

```
      .section .data, "aw", %progbits, unique, 1
      .word 1
      .section .data, "aw", %progbits, unique, 2
      .word 2
```

- Creating a section group containing two sections. Here, the G flag is used for the first section, using the `group_signature` symbol. The second section uses the ? flag to simplify making it part of the same group. Any further sections in this file using the G flag and `group_signature` symbol are placed in the same group.

```
      .section foo, "axG", %progbits, group_signature
get_value:
      movw r0, #:lower16:value
      movt r0, #:upper16:value
      ldr r0, [r0]
      bx lr

      .section bar, "aw?"
      .local value
value:
        .word 42
```

10.8 Conditional assembly directives

These directives allow you to conditionally assemble sequences of instructions and directives.

Syntax

```
.if[modifier] expression
    // ...
  [.elseif expression
    // ...]
  [.else
    // ...]
.endif
```

Operation

There are several different forms of the `.if` directive that check different conditions. Each `.if` directive must have a matching `.endif` directive. A `.if` directive can optionally have one associated `.else` directive, and can optionally have any number of `.elseif` directives.

You can nest these directives, with the maximum nesting depth limited only by the amount of memory in your computer.

The following forms if the `.if` directive are available, which check different conditions:

Table 10-17: .if condition modifiers

.if condition modifier	Meaning
`.if` *`expr`*	Assembles the following code if *`expr`* evaluates to non zero.
`.ifne` *`expr`*	Assembles the following code if *`expr`* evaluates to non zero.
`.ifeq` *`expr`*	Assembles the following code if *`expr`* evaluates to zero.
`.ifge` *`expr`*	Assembles the following code if *`expr`* evaluates to a value greater than or equal to zero.
`.ifle` *`expr`*	Assembles the following code if *`expr`* evaluates to a value less than or equal to zero.
`.ifgt` *`expr`*	Assembles the following code if *`expr`* evaluates to a value greater than zero.
`.iflt` *`expr`*	Assembles the following code if *`expr`* evaluates to a value less than zero.
`.ifb` *`text`*	Assembles the following code if the argument is blank.
`.ifnb` *`text`*	Assembles the following code if the argument is not blank.
`.ifc` *`string1 string2`*	Assembles the following code if the two strings are the same. The strings may be optionally surrounded by double quote characters ("). If the strings are not quoted, the first string ends at the first comma character, and the second string ends at the end of the statement (newline or semicolon).

.if condition modifier	Meaning
`.ifnc` *string1 string2*	Assembles the following code if the two strings are not the same. The strings may be optionally surrounded by double quote characters ("). If the strings are not quoted, the first string ends at the first comma character, and the second string ends at the end of the statement (newline or semicolon).
`.ifeqs` *string1 string2*	Assembles the following code if the two strings are the same. Both strings must be quoted.
`.ifnes` *string1 string2*	Assembles the following code if the two strings are not the same. Both strings must be quoted.
`.ifdef` *expr*	Assembles the following code if symbol was defined earlier in this file.
`.ifndef` *expr*	Assembles the following code if symbol was not defined earlier in this file.

The `.elseif` directive takes an expression argument but does not take a condition modifier, and therefore always behaves the same way as `.if`, assembling the subsequent code if the expression is not zero, and if no previous conditions in the same `.if .elseif` chain were true.

The `.else` directive takes no argument, and the subsequent block of code is assembled if none of the conditions in the same `.if .elseif` chain were true.

Examples

```
    // A macro to load an immediate value into a register. This expands to one or
    // two instructions, depending on the value of the immediate operand.
    .macro get_imm, reg, imm
      .if \imm >= 0x10000
        movw \reg, #\imm & 0xffff
        movt \reg, #\imm >> 16
      .else
        movw \reg, #\imm
      .endif
    .endm

    // The first of these macro invocations expands to one movw instruction,
    // the second expands to a movw and a movt instruction.
get_constants:
    get_imm r0, 42
    get_imm r1, 0x12345678
    bx lr
```

10.9 Macro directives

The `.macro` directive defines a new macro.

Syntax

```
    .macro macro_name [, parameter_name]...
      // ...
      [.exitm]
    .endm
```

Description

`macro_name`

The name of the macro.

`parameter_name`

Inside the body of a macro, the parameters can be referred to by their name, prefixed with `\`. When the macro is instantiated, parameter references are expanded to the value of the argument.

Parameters can be qualified in these ways:

Table 10-18: Macro parameter qualifier

Parameter qualifier	Meaning
`<`*`name`*`>:req`	This marks the parameter as required, it is an error to instantiate the macro with a blank value for this parameter.
`<`*`name`*`>:varag`	This parameter consumes all remaining arguments in the instantiation. If used, this must be the last parameter.
`<`*`name`*`>=<`*`value`*`>`	Sets the default value for the parameter. If the argument in the instantiation is not provided or left blank, then the default value is used.

Operation

The `.macro` directive defines a new macro with name *`macro_name`*, and zero or more named parameters. The body of the macro extends to the matching `.endm` directive.

Once a macro is defined, it can be instantiated by using it like an instruction mnemonic:

```
<macro_name> argument[, argument]...
```

Inside a macro body, `\@` expands to a counter value which is unique to each macro instantiation. This can be used to create unique label names, which do not interfere with other instantiations of the same macro.

The `.exitm` directive allows exiting a macro instantiation before reaching the end.

Examples

```
    // Macro for defining global variables, with the symbol binding, type and
    // size set appropriately. The 'value' parameter can be omitted, in which
    // case the variable gets an initial value of 0. It is an error to not
    // provide the 'name' argument.
    .macro global_int, name:req, value=0
    .global \name
    .type \name, %object
    .size \name, 4
\name:
    .word \value
    .endm

    .data
    global_int foo
    global_int bar, 42
```

10.10 Symbol binding directives

These directives modify the ELF binding of one or more symbols.

Syntax

```
    .global symbol[, symbol]...

    .local symbol[, symbol]...

    .weak symbol[, symbol]...
```

Description

`.global`

The `.global` directive sets the symbol binding to `STB_GLOBAL`. These symbols are visible to all object files being linked, so a definition in one object file can satisfy a reference in another.

`.local`

The `.local` directive sets the symbol binding in the symbol table to `STB_LOCAL`. These symbols are not visible outside the object file they are defined or referenced in, so multiple object files can use the same symbol names without interfering with each other.

`.weak`

The `.weak` directive sets the symbol binding to `STB_WEAK`. These symbols behave similarly to global symbols, with these differences:

- If a reference to a symbol with weak binding is not satisfied (no definition of the symbol is found), this is not an error.
- If multiple definitions of a weak symbol are present, this is not an error. If a definition of the symbol with strong binding is present, that one satisfies all references to the symbol, otherwise one of the weak references are chosen.

Operation

The symbol binding directive can be at any point in the assembly file, before or after any references or definitions of the symbol.

If the binding of a symbol is not specified using one of these directives, the default binding is:

- If a symbol is not defined in the assembly file, it has global visibility by default.
- If a symbol is defined in the assembly file, it has local visibility by default.

Note

`.local` and `.L` are different directives. Symbols starting with `.L` are not put into the symbol table.

Examples

```
  // This function has global binding, so can be referenced from other object
  // files. The symbol 'value' defaults to local binding, so other object
  // files can use the symbol name 'value' without interfering with this
  // definition and reference.
  .global get_val
get_val:
  ldr r0, value
  bx lr
value:
  .word 0x12345678

  // The symbol 'printf' is not defined in this file, so defaults to global
  // binding, so the linker searches other object files and libraries to
  // find a definition of it.
  bl printf

  // The debug_trace symbol is a weak reference. If a definition of it is
  // found by the linker, this call is relocated to point to it. If a
  // definition is not found (e.g. in a release build, which does not include
  // the debug code), the linker points the bl instruction at the next
  // instruction, so it has no effect.
  .weak debug_trace
  bl debug_trace
```

10.11 Org directive

The .org directive advances the location counter in the current section to new-location.

Syntax

```
.org new_location [, fill_value]
```

Description

new_location

The *new_location* argument must be one of:

- An absolute integer expression, in which case it is treated as the number of bytes from the start of the section.
- An expression which evaluates to a location in the current section. This could use a symbol in the current section, or the current location ('.').

fill_value

This is an optional 1-byte value.

Operation

The `.org` directive can only move the location counter forward, not backward.

By default, the `.org` directive inserts zero bytes in any locations that it skips over. This can be overridden using the optional *fill_value* argument, which sets the 1-byte value that is repeated in each skipped location.

Examples

```
  // Macro to create one AArch64 exception vector table entry. Each entry
  // must be 128 bytes in length. If the code is shorter than that, padding
  // is inserted. If the code is longer than that, the .org directive
  // reports an error, as this would require the location counter to move
  // backwards.
  .macro exc_tab_entry, num
1:
  mov x0, #\num
  b unhandled_exception
  .org 1b + 0x80
  .endm

  // Each of these macro instantiations emits 128 bytes of code and padding.
  .section vectors, "ax"
  exc_tab_entry 0
  exc_tab_entry 1
  // More table entries...
```

10.12 AArch32 Target selection directives

The AArch32 target selection directives specify code generation parameters for AArch32 targets.

Syntax

```
.arm

.thumb

.arch arch_name

.cpu cpu_name

.fpu fpu_name

.arch_extension extension_name

.eabi_attribute tag, value
```

Description

`.arm`

The `.arm` directive instructs the assembler to interpret subsequent instructions as A32 instructions, using the UAL syntax.

The `.code 32` directive is an alias for `.arm`.

`.thumb`

The `.thumb` directive instructs the assembler to interpret subsequent instructions as T32 instructions, using the UAL syntax.

The `.code 16` directive is an alias for `.thumb`.

.arch

The `.arch` directive changes the architecture that the assembler is generating instructions for. The `arch_name` argument accepts the same names as the `-march` option, but does not accept the optional architecture extensions accepted by the command-line option.

.cpu

The `.cpu` directive changes the CPU that the assembler is generating instructions for. The `cpu_name` argument accepts the same names as the `-mcpu` option, but does not accept the optional architecture extensions accepted by the command-line option.

.fpu

The `.fpu` directive changes the FPU that the assembler is generating instructions for. The `fpu_name` argument accepts the same names as the `-mfpu` option.

.arch_extension

The `.arch_extension` enables or disables optional extensions to the architecture or CPU that the assembler is generating instructions for. It accepts the following optional extensions, which can be prefixed with `no` to disable them:

- `crc`
- `fp16`
- `ras`

.eabi_attribute

The `.eabi_attribute` directive sets a build attribute in the output file. Build attributes are used by armlink to check for co-compatibility between object files, and to select suitable libraries.

The `.eabi_attribute` directive does not have any effect on which instructions the assembler accepts. It is recommended that the `.arch`, `.cpu`, `.fpu` and `.arch_extension` directives are used where possible, as they also check that no instructions not valid for the selected architecture are valid. These directives also set the relevant build attributes, so the `.eabi_attribute` directive is only needed for attributes not covered by them.

tag

The tag argument specifies the tag that is to be set. This can either be the tag name or tag number, but not both.

value

The value argument specifies the value to set for the *tag*. The value can either be of integer or string type. The type must match exactly the type expected for that tag.

Note

`Tag_compatibility` is a special tag that requires both an integer value and a string value:

```
.eabi_attribute Tag_compatibility, integer_value, string_value
```

Examples

```
// Generate code for the Armv7-M architecture:
.arch armv7-m

// Generate code for the Cortex-R5, without an FPU:
.cpu cortex-r5
.fpu none

// Generate code for Armv8.2-A with the FP16 extension:
.arch armv8.2-a
.fpu neon-fp-armv8
.arch_extension fp16
```

10.13 AArch64 Target selection directives

The AArch64 target selection directives specify code generation parameters for AArch64 targets.

Syntax

```
    .arch arch_name[+[no]extension]...

    .cpu cpu_name[+[no]extension]...
```

Description

`.arch`

The `.arch` directive changes the architecture that the assembler is generating instructions for.

The *`arch_name`* argument accepts the same names as the `-march` option, and accepts certain optional architecture extensions (*`extension`*) separated by `+`. The *`extension`* can be prefixed with `no` to disable it.

`.cpu`

The `.cpu` directive changes the CPU that the assembler is generating instructions for.

The *`cpu_name`* argument accepts the same names as the `-mcpu` option, and accepts certain optional architecture extensions (*`extension`*) separated by `+`. The *`extension`* can be prefixed with `no` to disable it.

`extension`

Optional architecture extensions. The accepted architecture extensions are:

- `crc`
- `crypto`
- `fp`
- `ras`
- `simd`

Examples

```
// Generate code for Armv8-A without a floating-point unit. The assembler
// reports an error if any instructions following this directive require
// the floating-point unit.
.arch armv8-a+nofp
```

10.14 Space-filling directives

The .space directive emits count bytes of data, each of which has value value. If the value argument is omitted, it defaults to zero.

Syntax

```
    .space count [, value]
```

```
    .fill count [, size [, value]]
```

Description

`.space`

The `.space` directive emits *`count`* bytes of data, each of which has value *`value`* . If the *`value`* argument is omitted, its default value is zero.

The `.skip` and `.zero` directives are aliases for the `.space` directive.

`.fill`

The `.fill` directive emits *`count`* data values, each with length *`size`* bytes and value *`value`* . If *`size`* is greater than 8, it is truncated to 8. If the *`size`* argument is omitted, its default value is one. If the *`value`* argument is omitted, its defaults value is zero.

The `.fill` directive always interprets the *`value`* argument as a 32-bit value.

- If the *`size`* argument is less than or equal to 4, the *`value`* argument is truncated to *`size`* bytes, and emitted with the appropriate endianness for the target. The assembler does not emit a diagnostic if *`value`* is truncated in this case.
- If the *`size`* argument is greater than 4, the value is emitted as a 4-byte value with the appropriate endianness. The value is emitted in the 4 bytes of the allocated memory with the lowest addresses. The remaining bytes in the allocated memory are then filled with zeroes. In this case, the assembler does emit a diagnostic if the value is truncated.

10.15 Type directive

The `.type` directive sets the type of a symbol.

Syntax

```
    .type symbol, %type
```

Description

`.type`

The `.type` directive sets the type of a symbol.

`symbol`

The symbol name to set the type for.

`%`*`type`*

The following types are accepted:

- `%function`
- `%object`
- `%tls_object`

Examples

```
  // 'func' is a function
  .type func, %function
func:
  bx lr

  // 'value' is a data object:
  .type value, %object
value:
  .word 42
```

11. armclang Inline Assembler

Provides reference information on writing inline assembly.

11.1 Inline Assembly

`armclang` provides an inline assembler that enables you to write assembly language sequences in C and C++ language source files. The inline assembler also provides access to features of the target processor that are not available from C or C++.

You can use inline assembly in two contexts:

- File-scope inline assembly statements.

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

- Inline assembly statement within a function.

```
void set_translation_table(void *table) {
    __asm("msr TTBR0_EL1, %0"
      :
      : "r" (table));
}
```

Both syntaxes accept assembly code as a string. Write your assembly code in the syntax that the integrated assembler accepts. For both syntaxes, the compiler inserts the contents of the string into the assembly code that it generates. All assembly directives that the integrated assembler accepts are available to use in inline assembly. However, the state of the assembler is not reset after each block of inline assembly. Therefore, avoid using directives in a way that affects the rest of the assembly file, for example by switching the instruction set between A32 and T32. See armclang Integrated Assembler.

Implications for inline assembly with optimizations

You can write complex inline assembly that appears to work at some optimization levels, but the assembly is not correct. The following examples describe some situations when inline assembly is not guaranteed to work:

- Including an instruction that generates a literal pool. There is no guarantee that the compiler can place the literal pool in the range of the instruction.
- Using or referencing a function only from the inline assembly without telling the compiler that it is used. The compiler treats the assembly as text. Therefore, the compiler can remove the function that results in an unresolved reference during linking. The removal of the function is especially visible for LTO, because LTO performs whole program optimization and is able to remove more functions.

For file-scope inline assembly, you can use the `__attribute((used))` function attribute to tell the compiler that a function is used. For inline assembly statements, use the input and output operands.

For large blocks of assembly code where the overhead of calling between C or C++ and assembly is not significant, Arm recommends using a separate assembly file, which does not have these limitations.

Related information

File-scope inline assembly on page 223
Output and input operands on page 225

11.2 File-scope inline assembly

Inline assembly can be used at file-scope to insert assembly into the output of the compiler.

All file-scope inline assembly code is inserted into the output of the compiler before the code for any functions or variables declared in the file, regardless of where they appear in the input. If multiple blocks of file-scope inline assembly code are present in one file, they are emitted in the same order as they appear in the source code.

Compiling multiple files containing file-scope inline assembly with the `-flto` option does not affect the ordering of the blocks within each file, but the ordering of blocks in different files is not defined.

Syntax

```
__asm("assembly code");
```

If you include multiple assembly statements in one file-scope inline assembly block, you must separate them by newlines or semicolons. The assembly string does not have to end in a new-line or semicolon.

Examples

```
// Simple file-scope inline assembly.
__asm(".global __use_realtime_heap");

// Multiple file-scope inline assembly statements in one block:
__asm("add_ints:\n"
    "  add r0, r0, r1\n"
    "  bx lr");

// C++11 raw string literals can be used for long blocks, without needing to
// include escaped newlines in the assembly string (requires C++11):
__asm(R"(
  sub_ints:
    sub r0, r0, r1
    bx lr
)");
```

11.3 Inline assembly statements within a function

Inline assembly statements can be used inside a function to insert assembly code into the body of a C or C++ function.

Inline assembly code allows for passing control-flow and values between C/C++ and assembly at a fine-grained level. The values that are used as inputs to and outputs from the assembly code must be listed. Special tokens in the assembly string are replaced with the registers that contain these values.

As with file-scope inline assembly, you can use any instructions or directives that are available in the integrated assembler in the assembly string. Use multiple assembly statements in the string of one inline assembly statement by separating them with newlines or semicolons. If you use multiple instructions in this way, the optimizer treats them as a complete unit. It does not split them up, reorder them, or omit some of them.

The compiler does not guarantee that the ordering of multiple inline assembly statements are preserved. It might also do the following:

- Merge two identical inline assembly statements into one inline assembly statement.
- Split one inline assembly statement into two inline assembly statements.
- Remove an inline assembly statement that has no apparent effect on the result of the program.

To prevent the compiler from doing any of these operations, you must use the input and output operands and the `volatile` keyword to indicate to the compiler which optimizations are valid.

The compiler does not parse the contents of the assembly string, except for replacing template strings, until code-generation is complete. It relies on the input and output operands, and clobbers to tell it about the requirements of the assembly code, and constraints on the surrounding generated code. Therefore the input and output operands, and clobbers must be accurate.

Syntax

```
__asm [volatile] (
    "<assembly string>"
    [ : <output operands>
    [ : <input operands>
    [ : <clobbers> ] ] ]
    );
```

11.3.1 Assembly string

An assembly string is a string literal that contains the assembly code.

The assembly string can contain template strings, starting with `%`, which the compiler replaces. The main use of these strings is to use registers that the compiler allocates to hold the input and output operands.

Syntax

Template strings for operands can take one of the following forms:

```
"%<modifier><number>"
"%<modifier>[<name>]"
```

`<modifier>` is an optional code that modifies the format of the operand in the final assembly string. You can use the same operand multiple times with different modifiers in one assembly string. See Inline assembly template modifiers.

For numbered template strings, the operands of the inline assembly statement are numbered, starting from zero, in the order they appear in the operand lists. Output operands appear before input operands.

If an operand has a name in the operand lists, you can use this name in the template string instead of the operand number. Square brackets must surround the name. Using names makes larger blocks of inline assembly easier to read and modify.

The `%%` template string emits a `%` character into the final assembly string.

The `%=` template string emits a number that is unique to the instance of the inline assembly statement. See Duplication of labels in inline assembly statements.

11.3.2 Output and input operands

The inline assembly statement can optionally accept two lists of operand specifiers, the first for outputs and the second for inputs. These lists are used to pass values between the assembly code and the enclosing C/C++ function.

Syntax

Each list is a comma-separated list of operand specifiers. Each operand specifier can take one of the following two forms:

```
[<name>] "<constraint>" (<value>)
         "<constraint>" (<value>)
```

Where:

`<name>`

Is a name for referring to the operand in templates inside the inline assembly string. If the name for an operand is omitted, it must be referred to by number instead.

`<constraint>`

Is a string that tells the compiler how the value is used in the assembly string, including:

- For output operands, whether it is only written to, or both read from and written to. Also whether it can be allocated to the same register as an input operand. See Constraint modifiers.

- Whether to store the value in a register or memory, or whether it is a compile-time constant. See Constraint codes.

`<value>`

Is a C/C++ value that the operand corresponds to. For output operands, this value must be a writable value.

Example

```
// foo.c

int saturating_add(int a, int b) {
    int result;
    __asm(
          // The assembly string uses templates for the registers which hold output
          // and input values. These are replaced with the names of the
          // registers that the compiler chooses to hold the output and input
          // values.

          "qadd %0, %[lhs], %[rhs]"

          // The output operand, which corresponds to the "result" variable. This
          // does not have a name assigned, so must be referred to in the assembly
          // string by its number ("%0").
          // The "=" character in the constraint string tells the compiler that the
          // register chosen to hold the result does not need to have any
          // particular value at the start of the inline assembly.
          // The "r" character in the constraint tells the compiler that the value
          // should be placed in a general-purpose register (r0-r12 or r14).

        : "=r" (result)

          // The two input operands also use the "r" character in their
          // constraints, so the compiler places them in general-purpose
          // registers.
          // These have names specified, which can be used to refer to them in
          // the assembly string ("%[lhs]" and "%[rhs]").

        : [lhs] "r" (a), [rhs] "r" (b)
    );

    return result;
}
```

Build this example with the following command:

```
armclang --target=arm-arm-none-eabi -march=armv7-a -O2 -c -S foo.c -o foo.s
```

The assembly language source file `foo.s` that is generated contains:

```
    .section    .text.saturating_add,"ax",%progbits
    .hidden saturating_add            @ -- Begin function saturating_add
    .globl  saturating_add
    .p2align    2
    .type   saturating_add,%function
    .code   32                        @ @saturating_add
saturating_add:
    .fnstart
@ %bb.0:                            @ %entry
    @APP
    qadd r0,r0,r1
    @NO_APP
    bx lr
```

```
.Lfunc_end0:
    .size saturating_add, .Lfunc_end0-saturating_add
    .cantunwind
    .fnend
```

In this example:

- The compiler places the C function `saturating_add()` in a section that is called `.text.saturating_add`.
- Within the body of the function, the compiler expands the inline assembly statement into the `qadd r0, r0, r1` instruction between the comments `@APP` and `@NO_APP`. In `-S` output, the compiler always places code that it expands from inline assembly statements within a function between a pair of `@APP` and `@NO_APP` comments.
- The compiler uses the general-purpose register R0 for:
    - The `int a` parameter of the `saturating_add()` function.
    - The inline assembly input operand `%[lhs]`.
    - The inline assembly output operand `%0`.
    - The return value of the `saturating_add()` function.
- The compiler uses the general-purpose register R1 for:
    - The `int b` parameter of the `saturating_add()` function.
    - The inline assembly input operand `%[rhs]`.

11.3.3 Clobber list

The clobber list is a comma-separated list of strings. Each string is the name of a register that the assembly code potentially modifies, but for which the final value is not important. To prevent the compiler from using a register for a template string in an inline assembly string, add the register to the clobber list.

For example, if a register holds a temporary value, include it in the clobber list. The compiler avoids using a register in this list as an input or output operand, or using it to store another value when the assembly code is executed.

In addition to register names, you can use two special names in the clobber list:

"memory"

This string tells the compiler that the assembly code might modify any memory, not just variables that are included in the output constraints.

"cc"

This string tells the compiler that the assembly code might modify any of the condition flags N, Z, C, or V. In AArch64 state, these condition flags are in the `NZCV` register. In AArch32 state, these condition flags are in the `CPSR` register.

Example

```
void enable_aarch64() {
    // Set bit 10 of SCR_EL3, to enale AArch64 at EL2.
    __asm volatile(R"(
          mrs x0, SCR_EL3
          orr x0, x0, #(1<<10)
          msr SCR_EL3, x0
          )" : /* no outputs */ : /* no inputs */
          // We used x0 as a temporary register, so we need to mark it as
          // clobbered, to prevent the compiler from storing a value in it.
        : "x0");
}
```

11.3.4 volatile

The optional `volatile` keyword tells the compiler that the assembly code has side-effects that the output, input, and clobber lists do not represent. For example, use this keyword with inline assembly code that sets the value of a system register.

The compiler assumes that any inline assembly statement with no output operands is `volatile`, even if the keyword is not present. However, Arm recommends that you still use it for clarity, and to avoid a behavior change if an output is added later.

Example

```
// Example where the volatile keyword is required. If the volatile keyword
// was omitted, this appears to still work. However, if the compiler were to
// inline it into a function that does not use the return value (old_table),
// then the inline assembly statement would appear to be unnecessary, and
// could get optimized out. The "volatile" keyword lets the compiler know
// that the assembly has an effect other than providing the output value, so
// that this does not happen.
void *swap_ttbr0(void *new_table) {
    void *old_table;
    __asm volatile (
          "mrs %[old], TTBR0_EL1\n"
          "msr TTBR0_EL1, %[new]\n"
        : [old] "=&r" (old_table)
        : [new] "r"   (new_table));
    return old_table;
}
```

11.4 Inline assembly constraint strings

A constraint string is a string literal, the contents of which are composed of two parts.

The contents of the constraint string are:

- A constraint modifier if the constraint string is for an output operand.
- One or more constraint codes.

11.4.1 Constraint modifiers

All output operands require a constraint modifier. There are currently no supported constraint modifiers for input operands.

Table 11-1: Constraint modifiers

Modifier	Meaning
`=`	This operand is only written to, and only after all input operands have been read for the last time. Therefore the compiler can allocate this operand and an input to the same register or memory location.
`+`	This operand is both read from and written to.
`=&`	This operand is only written to. It might be modified before the assembly block finishes reading the input operands. Therefore the compiler cannot use the same register to store this operand and an input operand. Operands with the `=&` constraint modifier are known as early-clobber operands. **Note:** In the case where a register constraint operand and a memory constraint operand are used together, you must use the `=&` constraint modifier on the register constraint operand to prevent the register from being used in the code generated to access the memory.

11.4.2 Constraint codes

Constraint codes define how to pass an operand between assembly code and the surrounding C or C++ code.

There are three categories of constraint codes:

Constant operands

You can only use these operands as input operands, and they must be compile-time constants. Use where a value is used as an immediate operand to an instruction. There are target-specific constraints that accept the immediate ranges suitable for different instructions.

Register operands

You can use these operands as both input and output operands. The compiler allocates a register to store the value. As there are a limited number of registers, it is possible to write an inline assembly statement for which there are not enough available registers. In this case, the compiler reports an error. The exact number of available registers varies depending on the target architecture and the optimization level.

Memory operands

You can use these operands as both input and output operands. Use them with load and store instructions. Usually a register is allocated to hold a pointer to the operand. As there are a limited number of registers, it is possible to write an inline assembly statement for which there are not enough available registers. In this case, the compiler reports an error. The exact number of available registers can vary depending on the target architecture and the optimization level.

There are some common constraints, which can be used in both AArch32 state and AArch64 state. Other constraints are specific to AArch32 state or AArch64 state. In AArch32 state, there are some constraints that vary depending on the selected instruction set.

11.4.3 Constraint codes common to AArch32 state and AArch64 state

The following constraint codes are common to both AArch32 state and AArch64 state.

Constants

i	A constant integer, or the address of a global variable or function.
n	A constant integer.

Note

The immediate constraints only check that their operand is constant after optimizations have been applied. Therefore it is possible to write code that you can only compile at higher optimization levels. Arm recommends that you test your code at multiple optimization levels to ensure it compiles.

Memory

m	A memory reference. This constraint causes a general-purpose register to be allocated to hold the address of the value instead of the value itself. By default, this register is printed as the name of the register surrounded by square brackets, suitable for use as a memory operand. For example, `[r4]` or `[x7]`. In AArch32 state only, you can print the register without the surrounding square brackets by using the `m` template modifier. See Template modifiers for AArch32 state.

Other

X	If the operand is a constant after optimizations have been performed, this constraint is equivalent to the `i` constraint. Otherwise, it is equivalent to the `r` or `w` constraints, depending on the type of the operand.

Note

Arm recommends that you use more precise constraints where possible. The `x` constraint does not perform any of the range checking or register restrictions that the other constraints perform.

11.4.4 Constraint codes for AArch32 state

The following constraint codes are specific to AArch32 state.

Registers

r

Operand must be an integer or floating-point type.

For targets that do not support Thumb®-2 technology, the compiler can use R0-R7.

For all other targets, the compiler can use R0-R12, or R14.

l

Operand must be an integer or floating-point type.

For T32 state, the compiler can use R0-R7.

For A32 state, the compiler can use R0-R12, or R14.

h

Operand must be an integer or floating-point type.

For T32 state, the compiler can use R8-R12, or R14.

Not valid for A32 state.

w

Operand must be a floating-point or vector type, or a 64-bit integer.

The compiler can use S0-S31, D0-D31, or Q0-Q15, depending on the size of the operand type.

t

Operand must be a 32-bit floating-point or integer type.

The compiler can use S0-S31.

The compiler never selects a register that is not available for register allocation. Similarly, R9 is reserved when compiling with `-frwpi`, and is not selected. The compiler may also reserve one or two registers to use as a frame pointer and a base pointer. The number of registers available for inline assembly operands therefore may be less than the number implied by the ranges above. This number may also vary with the optimization level.

If you use a 64-bit value as an operand to an inline assembly statement in A32 or 32-bit T32 instructions, and you use the `r` constraint code, then an even/odd pair of general purpose registers is allocated to hold it. This register allocation is not guaranteed for the `l` or `h` constraints.

Using the `r` constraint code enables the use of instructions like LDREXD/STREXD, which require an even/odd register pair. You can reference the registers holding the most and least significant halves of the value with the `Q` and `R` template modifiers. For an example of using template modifiers, see Template modifiers for AArch32 state.

Constants

The constant constraints accept different ranges depending on the selected instruction set. These ranges correspond to the ranges of immediate operands that are available for the different instruction sets. You can use them with a register constraint (see Using multiple alternative operand constraints) to write inline assembly that emits optimal code for multiple architectures without having to change the assembly code. The emitted code uses immediate operands when possible.

Constraint code	16-bit T32 instructions	32-bit T32 instructions	A32 instructions
`I`	[0, 255]	Modified immediate value for 32-bit T32 instructions.	Modified immediate value for A32 instructions.
`J`	[-255, -1]	[-4095, 4095]	[-4095, 4095]
`K`	8-bit value shifted left any amount.	Bitwise inverse of a modified immediate value for a 32-bit T32 instruction.	Bitwise inverse of a modified immediate value for an A32 instruction.
`L`	[-7, 7]	Arithmetic negation of a modified immediate value for a 32-bit T32 instruction.	Arithmetic negation of a modified immediate value for an A32 instruction.

11.4.5 Constraint codes for AArch64 state

The following constraint codes are specific to AArch64 state.

Registers

r
The compiler can use a 64-bit general purpose register, X0-X30.

If you want the compiler to use the 32-bit general purpose registers W0-W31 instead, use the `w` template modifier.

w
The compiler can use a SIMD or floating-point register, V0-V31.

The `b`, `h`, `s`, `d`, and `q` template modifiers can override this behavior.

x
Operand must be a 128-bit vector type.

The compiler can use a low SIMD register, V0-V15.

Constants

z
A constant with value zero, printed as the zero register (XZR or WZR). Useful when combined with `r` (see Using multiple alternative operand constraints) to represent an operand that can be either a general-purpose register or the zero register.

I
[0, 4095], with an optional left shift by 12. The range that the ADD and SUB instructions accept.

J
[-4095, 0], with an optional left shift by 12.

K	An immediate that is valid for 32-bit logical instructions. For example, AND, ORR, EOR.
L	An immediate that is valid for 64-bit logical instructions. For example, AND, ORR, EOR.
M	An immediate that is valid for a MOV instruction with a destination of a 32-bit register. Valid values are all values that the `K` constraint accepts, plus the values that the MOVZ, MOVN, and MOVK instructions accept.
N	An immediate that is valid for a MOV instruction with a destination of a 64-bit register. Valid values are all values that the `L` constraint accepts, plus the values that the MOVZ, MOVN, and MOVK instructions accept.

Related information

Inline assembly template modifiers on page 233

11.4.6 Using multiple alternative operand constraints

There are many instructions that can take either an immediate value with limited range or a register as one of their operands.

To generate optimal code for an instruction, use the immediate version of the instruction where possible. Using an immediate value avoids needing a register to hold the operand, and any extra instructions to load the operand into that register. However, you can only use an immediate value if the operand is a compile-time constant, and is in the appropriate range.

To generate the best possible code, you can provide multiple constraint codes for an operand. The compiler selects the most restrictive one that it can use.

Example

```
int add(int a, int b) {
    int r;
    // Here, the "Ir" constraint string tells the compiler that operand b can be
    // an immediate, but if it is not a constant, or not in the appropriate
    // range for an arithmetic instruction, it can be placed in a register.
    __asm("add %[r], %[a], %[b]"
        : [r] "=r" (r)
        : [a] "r" (a),
          [b] "Ir" (b));
    return r;
}

// At -O2 or above, the call to add is inlined and optimized, so that the
// immediate form of the add instruction can be used.
int add_42(int a) {
    return add(a, 42);
}

// Here, the immediate is not usable by the add instruction, so the compiler
// emits a movw instruction to load the value 12345 into a register.
int add_12345(int a) {
    return add(a, 12345);
}
```

11.5 Inline assembly template modifiers

Template modifiers are characters that you can insert into the assembly string, between the `%` character and the name or number of an operand reference. For example, `%c1`, where `c` is the template modifier, and `1` is the number of the operand reference. They change the way that the operand is printed in the string. This change is sometimes required so the operand is in the form that some instructions or directives expect.

11.5.1 Template modifiers common to AArch32 state and AArch64 state

Template modifiers that are common to both AArch32 state and AArch64 state.

These modiifiers are:

c

Valid for an immediate operand. Prints it as a plain value without a preceding `#`. Use this template modifier when using the operand in `.word`, or another data-generating directive, which needs an integer without the `#`.

n

Valid for an immediate operand. Prints the arithmetic negation of the value without a preceding `#`.

Example

```
// This uses an operand as the value in the .word directive. The .word
// directive does not accept numbers with a preceding #, so we use the 'c'
// template modifier to print just the value.
int foo() {
    int val;
    __asm (R"(
          ldr %0, 1f
          b 2f
        1:
          .word %c1
        2:
        )"
        : "=r" (val)
        : "i" (0x12345678));
    return val;
}
```

11.5.2 Template modifiers for AArch32 state

Template modifiers that are specific to AArch32 state.

These modiifiers are:

a	If the operand uses a register constraint, it is printed surrounded by square brackets. If it uses a constant constraint, it is printed as a plain immediate, with no preceding `#`.

y	The operand must be a 32-bit floating-point type, using a register constraint. It is printed as the equivalent D register with an index. For example, the register S2 is printed as `d1[0]`, and the register S3 is printed as `d1[1]`.
B	The operand must use a constant constraint, and is printed as the bitwise inverse of the value, without a preceding `#`.
L	The operand must use a constant constraint, and is printed as the least-significant 16 bits of the value, without a preceding `#`.
Q	The operand must use the `r` constraint, and must be a 64-bit integer or floating-point type. The operand is printed as the register holding the least-significant half of the value.
R	The operand must use the `r` constraint, and must be a 64-bit integer or floating-point type. The operand is printed as the register holding the most-significant half of the value.
H	The operand must use the `r` constraint, and must be a 64-bit integer or floating-point type. The operand is printed as the highest-numbered register holding half of the value.
e	The operand must be a 128-bit vector type, using the `w` or `x` constraint. The operand is printed as the D register that overlaps the low half of the allocated Q register.
f	The operand must be a 128-bit vector type, using the `w` or `x` constraint. The operand is printed as the D register that overlaps the high half of the allocated Q register.
m	The operand must use the `m` constraint, and is printed as a register without the surrounding square brackets.

Example

```
// In AArch32 state, the 'Q' and 'R' template modifiers can be used to print
// the registers holding the least- and most-significant half of a 64-bit
// operand.
uint64_t atomic_swap(uint64_t new_val, uint64_t *addr) {
    uint64_t old_val;
    unsigned temp;
    __asm volatile(
          "dmb ish\n"
          "1:\n"
          "ldrexd %Q[old], %R[old], %[addr]\n"
          "strexd %[temp], %Q[new], %R[new], %[addr]\n"
          "cmp %[temp], #0\n"
          "bne 1b\n"
          "dmb ish\n"
        : [new] "+&r" (old_val),
          [temp] "=&r" (temp)
        : [old] "r"   (new_val),
          [addr] "m"  (*addr));
    return old_val;
}
```

11.5.3 Template modifiers for AArch64 state

Template modifiers that are specific to AArch64 state.

In AArch64 state, register operands are printed as `x` registers for integer types and V registers for floating-point and vector types by default. You can use the template modifiers to override this behavior.

The modiifiers are:

w	Operand constraint must be `r`. Prints the register using its 32-bit `W` name.
x	Operand constraint must be `r`. Prints the register using its 64-bit `X` name.
b	Operand constraint must be `w` or `x`. Prints the register using its 8-bit `B` name.
h	Operand constraint must be `w` or `x`. Prints the register using its 16-bit `H` name.
s	Operand constraint must be `w` or `x`. Prints the register using its 32-bit `S` name.
d	Operand constraint must be `w` or `x`. Prints the register using its 64-bit `D` name.
q	Operand constraint must be `w` or `x`. Prints the register using its 128-bit `Q` name.

Example

```
// In AArch64 state, the 's' template modifiers cause these operands to be
// printed as S registers, instead of the default of V registers.
float add(float a, float b) {
    float result;
    __asm("fadd %s0, %s1, %s2"
        : "=w" (result)
        : "w"  (a), "w"  (b));
    return result;
}
```

11.6 Forcing inline assembly operands into specific registers

Sometimes specifying the exact register that is used for an operand is preferable to letting the compiler allocate a register automatically.

For example, the inline assembly block may contain a call to a function or system call that expects an argument or return value in a particular register.

To specify the register to use, the operand of the inline assembly statement must be a local register variable, which you declare as follows:

```
register int foo __asm("r2");
register float bar __asm("s4") = 3.141;
```

A local register variable is guaranteed to be held in the specified register in an inline assembly statement where it is used as an operand. Elsewhere it is treated as a normal variable, and can be stored in any register or in memory. Therefore a function can contain multiple local register variables that use the same register if only one local register variable is in any single inline assembly statement.

Example

```
// This function uses named register variables to make a Linux 'read' system call.
// The three arguments to the system call are held in r0-r2, and the system
// call number is placed in r7.
int syscall_read(register int fd, void *buf, unsigned count) {
    register unsigned r0 __asm("r0") = fd;
    register unsigned r1 __asm("r1") = buf;
    register unsigned r2 __asm("r2") = count;
    register unsigned r7 __asm("r7") = 0x900003;
    __asm("svc #0"
        : "+r" (r0)
        : "r"  (r1), "r" (r2), "r" (r7));
    return r0;
}
```

11.7 Symbol references and branches into and out of inline assembly

Symbols that are defined in an inline assembly statement can only be referred to from the same inline assembly statement.

The compiler can optimize functions containing inline assembly, which can result in the removal or duplication of the inline assembly statements. To define symbols in assembly and use them elsewhere, use file-scope inline assembly, or a separate assembly language source file.

Except for function calls, it is not permitted to branch out of an inline assembly block, including branching to other inline assembly blocks. The optimization passes of the compiler assume that inline assembly statements only exit by reaching the end of the assembly block, and optimize the surrounding function accordingly.

It is valid to call a function from inside inline assembly, as that function returns control-flow back to the inline assembly code.

Arm does not recommend directly referencing global data or functions from inside an assembly block by using their names in the assembly string. Often such references appear to work, but the compiler does not know about the reference.

If the global data or functions are only referenced inside inline assembly statements, then the compiler might remove these global data or functions.

To prevent the compiler from removing global data or functions which are referenced from inline assembly statements, you can:

- use `__attribute__((used))` with the global data or functions.
- pass the reference to global data or functions as operands to inline assembly statements.

Arm recommends passing the reference to global data or functions as operands to inline assembly statements so that if the final image does not require the inline assembly statements and the referenced global data or function, then they can be removed.

Example

```
static void foo(void) { /* ... */ }
// This function attempts to call the function foo from inside inline assembly.
// In some situations this may appear to work, but if foo is not referenced
// anywhere else (including if all calls to it from C got inlined), the
// compiler could remove the definition of foo, so this would fail to link.
void bar() {
    __asm volatile(
            "bl foo"
        : /* no outputs */
        : /* no inputs */
        : "r0", "r1", "r2", "r3", "r12", "lr");
}

// This function is the same as above, except it passes a reference to foo into
// the inline assembly as an operand. This lets the compiler know about the
// reference, so the definition of foo is not removed (unless, the
// definition of bar_fixed can also be removed). In C++, this has the
// additional advantage that the operand uses the source name of the function,
// not the mangled name (_ZL3foov) which would have to be used if writing the
// symbol name directly in the assembly string.
void bar_fixed() {
    __asm volatile(
            "bl %[foo]"
        : /* no outputs */
        : [foo] "i" (foo)
        : "r0", "r1", "r2", "r3", "r12", "lr");
}
```

11.8 Duplication of labels in inline assembly statements

You can use labels inside inline assembly, for example as the targets of branches or PC-relative load instructions. However, you must ensure that the labels that you create are valid if the compiler removes or duplicates an inline assembly statement.

Duplication can happen when a function containing an inline assembly statement is inlined in multiple locations. Removal can happen if an inline assembly statement is not reachable, or its result is unused and it has no side-effects.

If regular labels are used inside inline assembly, then duplication of the assembly could lead to multiple definitions of the same symbol, which is invalid. Multiple definitions can be avoided either by using numeric local labels , or using the `%=` template string. The `%=` template string is expanded to a number that is unique to each instance of an inline assembly statement. Duplicated statements have different numbers. All uses of `%=` in an instance of the inline assembly statement have the same value. You can therefore create label names that can be referenced in the same inline assembly statement, but which do not conflict with other copies of the same statement.

Note

Unique numbers from the `%=` template string might still result in the creation of duplicate labels. For example, label names `loop%=` and `loop1%=` can result in duplicate labels. The label for instance number 0 of `loop1%=` evaluates to `loop10`. The label for instance number 10 of `loop%=` also evaluates to `loop10`.

To avoid such duplicate labels, choose the label names carefully.

Example

```
void memcpy_words(int *src, int *dst, int len) {
    assert((len % 4) == 0);
    int tmp;
    // This uses the "%=" template string to create a label which can be used
    // elsewhere inside the assembly block, but which does not conflict with
    // inlined copies of it.
    // R is a C++11 raw string literal.
    __asm(R"(
        .Lloop%=:
            ldr %[tmp], %[src], #4
            str %[tmp], %[dst], #4
            subs %[len], #4
            bne .Lloop%=)"
        : [dst] "=&m" (*dst),
          [tmp] "=&r" (tmp),
          [len] "+r" (len)
        : [src] "m" (*src));
}
```

See the example in File-scope inline assembly.

12. armclang Reference Guide Changes

Describes the technical changes that have been made to the armclang Reference Guide.

12.1 Changes for the armclang Reference Guide

Changes that have been made to the *armclang Reference Guide* are listed with the latest version first.

Table 12-1: Changes between 6.6.5 (revision L) and 6.6.4 (revision K)

Change	Topics affected
[SDCOMP-56827] Changed Restrictions section title to Post-conditions, and slightly reworded the text.	• -ffunction-sections, -fno-function-sections.
[SDCOMP-57610] Added a description for the `armclang` options `-faggressive-jump-threading` and `-fno-aggressive-jump-threading`.	• -faggressive-jump-threading, -fno-aggressive-jump-threading. • Summary of armclang command-line options.
[SDCOMP-58966] Added note about using attribute **UNDEFINED** for an undefined instruction handler.	• __attribute__((interrupt("type"))) function attribute.
[SDCOMP-59492] Corrected `long double` IEEE precision statements for AArch64.	• Architecture. • Floating-point. • Basic concepts. • Expressions. • Support level definitions.
[SDCOMP-57521] Added a note that `armclang` always applies the rules for type auto-deduction from C++17, regardless of which C++ source language mode a program is compiled for.	• -std
[SDCOMP-57264] Added note on mixing objects compiled with different C/C++ standards.	• -std.
[SDCOMP-57811] Corrected the IEEE compliance statements for `fz` libraries.	• -ffast-math, -fno-fast-math. • -ffp-mode.
[SDCOMP-62125] Corrected the information for when the `__SOFTFP__` predefined macro is defined.	• Predefined macros.
Added the syntax for `-Rpass-missed` as a [COMMUNITY] option.	• -Rpass

Table 12-2: Changes between 6.6.4 (revision K) and 6.6.3 (revision J)

Change	Topics affected
[SDCOMP-54472] The note no longer states that a warning is emitted when using `-mexecute-only` with `-flto`.	• -flto, -fno-lto. • -mexecute-only. • -O (armclang).
[SDCOMP-54519] Modified the text for `-fno-fast-math` and corrected the IEEE compliance statements in the table.	• -ffast-math, -fno-fast-math.
[SDCOMP-54519] Modified the description of `full` and corrected the IEEE compliance statements in the table.	• -ffp-mode.

Change	Topics affected
[SDCOMP-51119] Removed the text that states the compiler issues a warning for priority values up to and including 100.	• __attribute__((constructor(priority))) function attribute.
[SDCOMP-54804] Added the *Volatile variables* topic.	• Volatile variables.