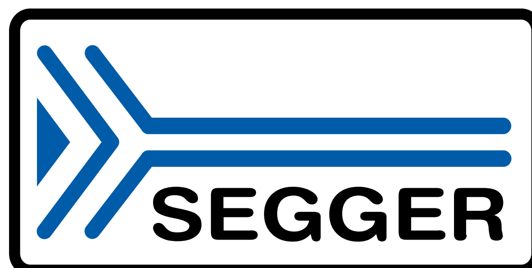
SEGGER Compiler

A C/C++ Compiler for ARM
and RISC-V microcontrollers

User Guide & Reference Manual

Document: UM20007
Software Version: 18.1.0
Revision: 0
Date: May 24, 2024



A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of SEGGER. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2019-2024 SEGGER Microcontroller GmbH, Monheim am Rhein / Germany

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel. +49 2173-99312-0
Fax. +49 2173-99312-28
E-mail: support@segger.com*
Internet: www.segger.com

Print date: May 24, 2024

Software	Date	By	Description
18.1.0	2024-02-05	RH	Initial release.

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The C/C++ programming language.
- The build process to create embedded applications.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
keyword	Command line keyword.
`<option>`	Placeholder for a parameter.
`Sample`	Sample code in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
SEGGER home page	A hyperlink to an external document or web site.

© 2019-2024 SEGGER Microcontroller GmbH

 © 2019-2024 SEGGER Microcontroller GmbH

Table of contents

1 About the compiler 7

1.1 Introduction 8
1.1.1 What is the SEGGER Compiler? 8
1.1.2 Compiler input and output files 8
1.2 Libraries 10

2 Running the Segger Compiler 11

2.1 General compiler invocation 12
2.2 Mandatory compiler options 13
2.2.1 Specifying the target architecture 13
2.2.2 Specifying target features 14
2.2.3 Specifying an optimization level 14
2.2.4 Specifying a language standard 14
2.2.5 Selecting floating-point options 15
2.2.5.1 Floating-point Application Binary Interface 15
2.2.5.2 Disabling floating-point hardware 16
2.2.6 C++ exceptions 16
2.2.7 Debugging 16
2.3 Controlling diagnostic messages 17
2.4 Useful options for embedded applications 18
2.4.1 Data type of enum 18
2.5 Running the compiler on assembly files 19

3 Compiler-specific extensions 20

3.1 Function and Variable Attributes 21

4 Controlling code generation 22

4.1 Unaligned memory access 23
4.1.1 Disable unaligned memory access optimizations 23
4.1.2 Special cases 23
4.2 Using the volatile keyword 25
4.3 Function inlining 27
4.3.1 Controlling inlining 27
4.4 Undefined behavior 28

5 Link Time Optimization 29

5.1 Typical build process with LTO 30
5.2 Compiler invocation for LTO 32

 © 2019-2024 SEGGER Microcontroller GmbH

6 Compiler options reference 33
6.1 Undocumented compiler features 34
6.2 List of all compiler options 35
6.3 Predefined macros 39
6.3.1 Predefined macros for ARM targets 39
6.3.2 Predefined macros for RISC-V targets 40
6.4 Complete list of supported target features 41
6.5 Implicit applied target features for ARM CPUs 45

 © 2019-2024 SEGGER Microcontroller GmbH

Chapter 1

About the compiler

© 2019-2024 SEGGER Microcontroller GmbH

1.1 Introduction

This section presents an overview of the SEGGER Compiler and its capabilities.

1.1.1 What is the SEGGER Compiler?

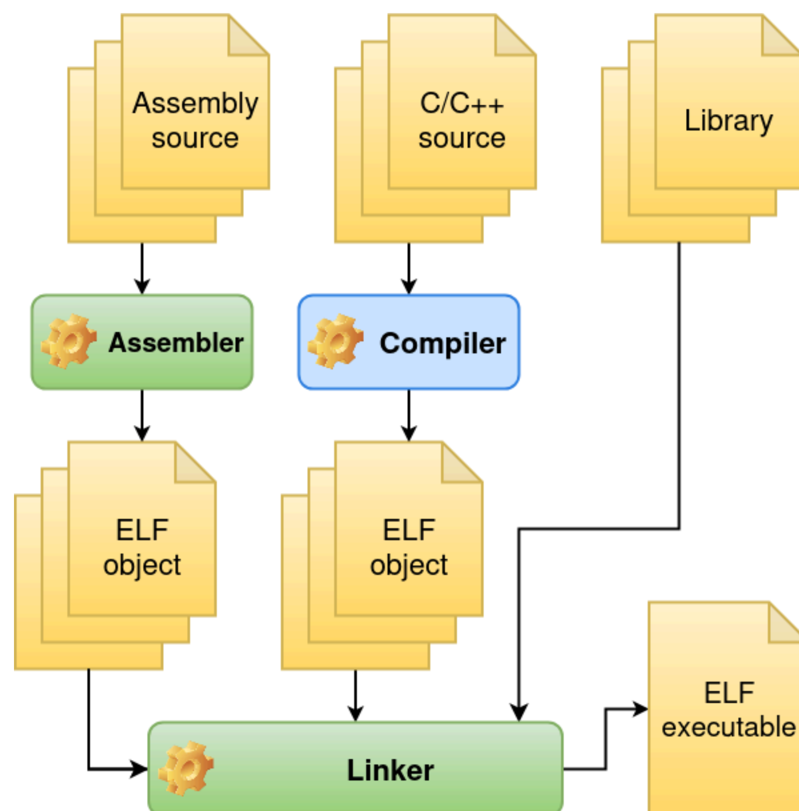
The SEGGER Compiler is an optimizing C/C++ compiler for ARM and RISC-V microcontrollers. It is based on The LLVM Compiler Infrastructure and Clang technology. Clang is a compiler front end for LLVM that supports the C and C++ programming languages.

The SEGGER Compiler shares the front-end with the clang compiler and therefore supports the latest C and C++ language features. The back end, which produces the binary objects for the target architecture has been optimized by SEGGER to generate fast and small Thumb/Thumb-2 code.

1.1.2 Compiler input and output files

The compiler usually translates C or C++ source code files into ELF object files. The resulting ELF object files are not executable on any CPU, they must be processed by a linker in order to create an executable file.

A typical build process looks like:



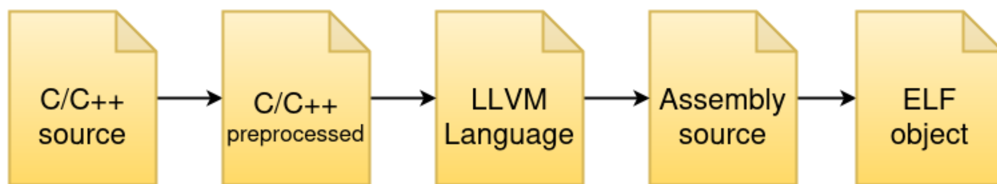
Other types of files can be generated and / or processed by the compiler, too. The following types of files are known by the compiler:

Type of file	filename extension	Input / Output
C-source. C source code according any ISO C standard.	`.c`	In
C++-source. C++ source code according any ISO C++ standard.	`.cpp .cc`	In
Header. Can be included by C/C++ source files. Not used as direct input to the compiler.	usually `.h`	

 © 2019-2024 SEGGER Microcontroller GmbH

Type of file	filename extension	Input / Output
C-preprocessed. C source processed by the C preprocessor.	`.i`	In+Out
C++-preprocessed. C++ source processed by the C preprocessor.	`.ii`	In+Out
LLVM Language. File containing LLVM's device independent code representation in binary format. It may be used as input for the LTO tool, see *Link Time Optimization*.	`.bc`	Out
Assembly. File containing assembly instructions for the given target processor.	`.asm .s`	In+Out
Object. Binary object file in ELF format. Not executable, must be processed by a linker to create an executable file.	`.o`	Out

When the compiler translates a C file into an object file, then internally the following translations are performed:



The intermediate results are usually not output to a file, but the compiler can be instructed to output any of the intermediate files instead of the object file and stop translation at this point. The following table shows all translations possible.

Input \ Output	*C/C++-preprocessed*	*LLVM Language*	*Assembly*	*Object*
C/C++-source	✓	✓	✓	✓
C/C++-preprocessed		✓	✓	✓
LLVM Language			✓	✓
Assembly				✓

 © 2019-2024 SEGGER Microcontroller GmbH

1.2 Libraries

In order to get an executable application the object files generated by the compiler must be linked against appropriate C/C++ runtime libraries. These libraries must provide:

- An implementation of the library features as defined in the C standards.
- An implementation of the ISO C++ library standard.
- An implementations of low-level language features.

 © 2019-2024 SEGGER Microcontroller GmbH

Chapter 2

Running the Segger Compiler

© 2019-2024 SEGGER Microcontroller GmbH

2.1 General compiler invocation

On each execution of the compiler only a single C/C++ input file can be processed and converted into an output file. To process assembly files, see *Running the compiler on assembly files*. General command line syntax:

```
segger-cc <options> <output-type> <input-file> -o <output-file>
```

`<output-type>` determines the type of output file to be generated by the compiler:

`<output-type>`	Type of output file
-c **-emit-obj**	Binary object file in ELF format
-S	Assembly file
-E	Preprocessed C/C++
-emit-llvm-bc	Binary LLVM Language file

The type of an input file is determined by the file extension, as described in *Compiler input and output files*. If the input file type doesn't match the file's extension, then the input file type to be used by the compiler can be specified with the **-x** option. For example: To compile a file `test.c` as C++ source, add the option:

```
-x c++
```

 © 2019-2024 SEGGER Microcontroller GmbH

2.2 Mandatory compiler options

2.2.1 Specifying the target architecture

It's required to specify the target architecture or target CPU for which the compiler shall generate code, using the option:

```
--target=<target-architecture>
```

or

```
--target=<target-cpu>
```

Supported architectures and cpus are:

ARM (32-bit) <target-architecture>

- armv4, armv4t, armv5t, armv5te, armv5tej
- armv6, armv6-m, armv6k, armv6kz, armv6t2
- armv7-a, armv7-m, armv7-r, armv7e-m, armv7ve
- armv8-a, armv8-m.base, armv8-m.main, armv8-r, armv8.1-a
- armv8.1-m.main, armv8.2-a, armv8.3-a, armv8.4-a, armv8.5-a
- armv8.6-a, armv8.7-a, armv8.8-a, armv8.9-a, armv9-a
- armv9.1-a, armv9.2-a, armv9.3-a, armv9.4-a, armv9.5-a

ARM (32-bit) <target-cpu>

- arm-arm1136j-s, arm-arm1136jf-s, arm-arm1156t2-s, arm-arm1156t2f-s
- arm-arm1176jz-s, arm-arm1176jzf-s, arm-arm710t, arm-arm720t
- arm-arm7tdmi, arm-arm7tdmi-s, arm-arm8, arm-arm810
- arm-arm9, arm-arm920, arm-arm920t, arm-arm922t
- arm-arm926ej-s, arm-arm940t, arm-arm946e-s, arm-arm966e-s
- arm-arm968e-s, arm-arm9e, arm-arm9tdmi, arm-cortex-a12
- arm-cortex-a15, arm-cortex-a17, arm-cortex-a32, arm-cortex-a35
- arm-cortex-a5, arm-cortex-a53, arm-cortex-a55, arm-cortex-a57
- arm-cortex-a7, arm-cortex-a710, arm-cortex-a72, arm-cortex-a73
- arm-cortex-a75, arm-cortex-a76, arm-cortex-a76ae, arm-cortex-a77
- arm-cortex-a78, arm-cortex-a78c, arm-cortex-a8, arm-cortex-a9
- arm-cortex-m0, arm-cortex-m0plus, arm-cortex-m1, arm-cortex-m23
- arm-cortex-m3, arm-cortex-m33, arm-cortex-m35p, arm-cortex-m4
- arm-cortex-m52, arm-cortex-m55, arm-cortex-m7, arm-cortex-m85
- arm-cortex-r4, arm-cortex-r4f, arm-cortex-r5, arm-cortex-r52
- arm-cortex-r7, arm-cortex-r8, arm-cortex-x1, arm-cortex-x1c

RISCV <target-architecture>

- riscv32
- riscv64

Instruction set for ARM targets

For 32-bit ARM targets it should also be specified if either the THUMB or ARM instruction set shall be used:

```
-marm
-mno_thumb
```

or

```
-mthumb
```

Default is THUMB instruction set, if supported by the architecture.

 © 2019-2024 SEGGER Microcontroller GmbH

2.2.2 Specifying target features

Each target architecture has a set of features that can be separately enabled or disabled. Target features include for example:

- Instruction set extensions like floating point, vector operation, DSP instructions
- Size of register sets
- Floating point precision
- Optimization preferences
- Capability for unaligned memory accesses

For each target CPU there is a predefined set of target features that are enabled by default, see *Implicit applied target features for ARM CPUs*. If one of these default target features shall not be used, it must be explicitly disabled.

To explicitly enable or disable a target feature, use the compiler option:

-target-feature +<feature-name> (enable)

or

-target-feature -<feature-name> (disable)

The compiler will output a list of actually applied target features for a compile run, when specifying:

-target-feature +list

See *Complete list of supported target features*.

2.2.3 Specifying an optimization level

It's recommended to always specify a suitable optimization level. The compiler supports these optimization levels:

Option	Optimization goals
-O0	No optimization. Generate object code that directly corresponds to the source code for best debugging experience.
-O1	Basic optimization. Still good debug experience.
-Os	Optimization balancing code size reduction and fast performance. Best suited for embedded applications.
-O2	Optimization for fast performance while accepting greater code size.
-Oz	Optimization for minimal code size while accepting slower execution speed.
-O3	More aggressive optimization for execution speed.

2.2.4 Specifying a language standard

Always specify a language standard to compile for, using the option:

-std=<standard>

The Compiler supports Standard and GNU variants of source languages as shown in the following table.

Language	Supported ISO standards	Supported GNU standards
C	c90 c99 c11 c17	gnu90 gnu99 gnu11 gnu17
C++	c++98 c++11 c++14 c++17 c++20 c++23	gnu++98 gnu++11 gnu++14 gnu++17 gnu++20 gnu++23

Because the compiler uses available language extensions by default, it does not check for strict ISO standard when selecting an ISO C/C++ standard. To compile to strict ISO standard for the source language, use the **-Wpedantic** or **-Wpedantic-erros** option. This options generate warnings or errors where the source code violates the ISO standard.

 © 2019-2024 SEGGER Microcontroller GmbH

2.2.5 Selecting floating-point options

The compiler supports floating-point arithmetic and floating-point data types by either

- Libraries that implement floating-point arithmetic in software, or
- Hardware floating-point registers and instructions that are available on most CPUs.

Code that uses floating-point hardware is more compact and faster than code that uses software libraries for floating-point arithmetic, but can only be run on processors that have the floating-point hardware. Code that uses software floating-point libraries can run on all processors, even on processors that do not have any floating-point hardware. Therefore, using software floating-point libraries makes the code more portable.

There are various options that determine how the compiler generates code for floating-point arithmetic. Depending on your target, one or more of these options may be required to generate floating-point code that correctly uses floating-point hardware or software libraries.

Code generation for floating-point arithmetic is mainly controlled by target features (see *Specifying target features*). After selecting a target architecture or target CPU then a set of floating-point options are enabled by default that match the capabilities of the hardware, see *Implicit applied target features for ARM CPUs*. If different than the default floating-point features shall be used by the compiler all of the default enabled features that shall not be used must be explicitly disabled. To disable all default floating-point features the option:

```
-target-feature -implicit-fp
```

can be specified. In this case only the floating point features explicitly enabled on the command line will be active.

2.2.5.1 Floating-point Application Binary Interface

Floating-point Application Binary Interface (ABI) refers to how the floating-point arguments are passed to and returned from function calls. The compiler can use hardware linkage or software linkage. The compiler passes and returns floating-point values either in general-purpose registers or in floating-point registers. Options to select the floating-point ABI for ARM targets:

Option	Behavior
-msoft-float **-mfloat-abi soft**	No use of any floating point unit. Calls to the C library are generated to implement floating point operations.
-mhard-float **-mfloat-abi hard**	Hardware floating-point instructions are generated to implement floating point operations and floating-point registers are used to pass floating point parameters on function calls.
-mfloat-abi softfp	Hardware floating-point instructions are generated to implement floating point operations, but general-purpose registers are used to pass floating point parameters on function calls.

Options to select the floating-point ABI for RISC-V targets:

Option	Behavior
-target-abi ilp32	Floating point arguments are passed in general purpose registers. No requirements on instruction set / hardware.
-target-abi ilp32f	32bit and smaller floating point types are passed in floating-point registers. Requires F type floating point registers and instructions (target feature **+f**).
-target-abi ilp32d	64bit and smaller floating point types are passed in floating-point registers. Requires D type floating point registers and instructions (target feature **+d**).

 © 2019-2024 SEGGER Microcontroller GmbH

2.2.5.2 Disabling floating-point hardware

To completely disable the use of floating-point hardware instructions for ARM targets use the options:

```
-msoft-float -target-feature -implicit-fp
```

Disabling floating-point arithmetic may not disable the use of the floating-point hardware completely because the floating-point hardware unit may still be used for special integer arithmetic operations. For example see target features `mve` and `mve.fp`.

2.2.6 C++ exceptions

In order to compile C++ source files that use exception handling (`throw()` and `catch()`), this must be enabled with the option:

```
-fcxx-exception
```

2.2.7 Debugging

In order to generate and store information for debugging into the output file, use the options:

```
-debug-info-kind=standalone -dwarf-version=4 -debugger-tuning=gdb
```

This generates debug information suitable for gdb or compatible debuggers. DWARF versions 2, 3, 4, or 5 are supported.

> **Note**
>
> Higher optimization levels usually result in poor correspondence of the generated code to the source code, which makes debugging more difficult. Therefore we recommend optimization levels `-O0` or `-O1` for debugging.

 © 2019-2024 SEGGER Microcontroller GmbH

2.3 Controlling diagnostic messages

The compiler provides diagnostic messages in the form of warnings and errors. The format of a diagnostic messages is:

```
<file>:<line>:<column>: <message-type>: <message-text>
```

Field	Contents
<file>	The name of the source file that contains the error or warning
<line>	The line number that contains the error or warning.
<column>	The column position of the error or warning.
<message-type>	The type of the message: **error**, **warning** or **note**.
<message-text>	The message text. This text might end with a diagnostic group of the form **-W**<group> to identify the diagnostic group that the message belongs to. The group can be used to suppress the message or change it's message type, see below.

If any **error** message is generated, the compiler will not create any output file and will return a non-zero exit status.

You can use options to suppress these messages or enable them as either warnings or errors:

Option	Description
-W<group>	Enable warnings of diagnostic group <group>
-Wall	This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid
-Wdeprecated	Enable warnings for deprecated constructs and define __DEPRECATED
-Werror	Turn all warnings into errors
-Werror=<group>	Turn all messages of diagnostic group <group> into errors
-Wextra	This enables some extra warning messages that are not enabled by -Wall
-Wno-<group>	Disable warnings of diagnostic group <group>
-Wno-error=<group>	Turn all messages of diagnostic group <group> into warnings
-Wpedantic	Issue warning when violations of the selected language standard are detected
-Wpedantic-errors	Issue errors when violations of the selected language standard are detected

Example

To suppress the warning:

```
File.c:557:10: warning: no previous prototype for 'foo' [-Wmissing-prototypes]
```

you can specify the option:

```
-Wno-missing-prototypes
```

 © 2019-2024 SEGGER Microcontroller GmbH

2.4 Useful options for embedded applications

On Embedded systems resources like memory and cpu time are usually restricted, therefore code and data size as well as execution speed are generally an issue. This section describes some compiler options that are strongly recommended or at least very useful to create efficient embedded application software.

2.4.1 Data type of enum

According to the C/C++ standard, an enumeration variable has underlying 'int' data type. But enumerations do not need to be stored in integers. If the range of values fits into a smaller data type, the compiler can use this instead.

Recommendation: Use option `-fshort-enums` if possible.

© 2019-2024 SEGGER Microcontroller GmbH

2.5 Running the compiler on assembly files

To compile assembly files the compiler contains an integrated assembler. A different invocation of the compiler is required:

```
segger-cc -cc1as <options> <assembly-file> -o <output-file>
```

The options available for the assembler are a subset of the compiler options, see *List of all compiler options*.

Mandatory options to compile assembly files

- *Specifying the target architecture*
- *Specifying target features*
- Options to generate and store information for *Debugging*

© 2019-2024 SEGGER Microcontroller GmbH

Chapter 3

Compiler-specific extensions

This chapter describes compiler extensions to the C and C++ Standards.

 © 2019-2024 SEGGER Microcontroller GmbH

3.1 Function and Variable Attributes

The SEGGER compiler provides function and variable attributes that are extensions to the C and C++ Standards. Attributes use the following syntax:

`__attribute__((<option+parameter>))`

Attributes can be placed before or after a function or variable declaration or definition, except for function definitions which must follow the attribute. Examples:

```
uint8_t Buff[1024] __attribute__((aligned(16)));

int foo1(unsigned x) __attribute__((section("my_text")));

__attribute__((section("my_text"))) int foo2(unsigned x, unsigned y) {
  return x+y;
}
```

Supported attributes:

Attribute	Applicable	Description
`__attribute__((section("<name>")))`	Variable, Function	Place the code or variable into the different section `"name"` of the image rather than `.text` or `.data`
`__attribute__((aligned(<n>)))`	Variable	Specifies a minimum alignment for the variable of `<n>` bytes. `<n>` must be a power of 2.
`__attribute__((always_inline))`	Function	Instruct the compiler to inline the function whenever possible
`__attribute__((noinline))`	Function	Prevent the inlining of a function
`__attribute__((noreturn))`	Function	Asserts that a function never returns.
`__attribute__((weak))`	Variable, Function	Export the function / variable symbol weakly.

 © 2019-2024 SEGGER Microcontroller GmbH

Chapter 4

Controlling code generation

© 2019-2024 SEGGER Microcontroller GmbH

4.1 Unaligned memory access

Target CPUs read and write memory more efficiently when they store data at an address that's a multiple of the data size. For example, a 4-byte integer is accessed more efficiently if it's stored at an address that's a multiple of 4. When data isn't aligned, an unaligned memory access may be required to access the data.

Some CPUs allow unaligned memory accesses, but need more clock cycles to access the data. Other CPUs are not capable of unaligned memory accesses and will either generate an exception or just don't execute the data access correctly.

As the compiler generally aligns data on natural boundaries that are based on the target processor and the size of the data, unaligned accesses are not required. But during optimization the compiler may generate unaligned memory accesses to improve code size and execution speed.

Example 1

```
uint32_t LoadU32LE(const uint8_t * pData) {
  uint32_t r;
  r  = *pData++;
  r |= (uint32_t)*pData++ << 8;
  r |= (uint32_t)*pData++ << 16;
  r |= (uint32_t)*pData   << 24;
  return r;
}
```

Example 2

```
struct Data_t { unsigned short a,b; };

void Init(struct Data_t *p) {
  p->a = 0;
  p->b = 1;
}
```

The compiler may generate a single 32-bit load instead of four times loading a byte for example 1, or a single 32-bit store instead of two 16-bit stores for example 2, even if the addressed 32-bit word is not aligned on a 4-byte boundary.

4.1.1 Disable unaligned memory access optimizations

This kind of optimizations are performed only for target processors which are known to support unaligned memory access. It can be disabled anyway, using the compiler option:

-target-feature +strict-align (for ARM targets)

-target-feature -fast-unaligned-access (for RISC-V targets)

It can also be disabled by declaring the access 'volatile':

```
uint32_t LoadU32LE(volatile const uint8_t * pData) {
  uint32_t r;
  r  = *pData++;
  ...
}
```

4.1.2 Special cases

Usually the compiled application works as expected and there is no need to care about unaligned memory accesses, but there are some exceptions to consider:

 © 2019-2024 SEGGER Microcontroller GmbH

Runtime configuration of unaligned memory support

On some processors the support for unaligned memory access can be enabled or disabled at runtime by setting a control register of the CPU. In this case either

- make sure that unaligned memory access is enabled in the CPU at startup of the application or
- disable generating of unaligned memory access using the compiler option above.

Special memory areas

Although unaligned memory access is supported by the target processor there may be certain memory areas in the target system where an unaligned memory access is not possible. In this case either

- protect all accesses to this special memory areas with the 'volatile' keyword in your source code or
- disable generating of unaligned memory access using the compiler option above.

Unaligned memory access caused by the source code

```
uint32_t LoadU32LE(const uint8_t * pData) {
  return *((const uint32_t *)pData);
}
```

The cast tells the compiler to access the memory location like a normal (aligned) 32-bit value which may result in an unaligned access. In general this is non portable code. Any compiler option or 'volatile' specifier will not prevent an unaligned access. To avoid unaligned access in this case:

- Make sure that the pointer `pData` is always correctly aligned before calling the function or
- write portable code avoiding nasty casts.

Note: The code above also depends on the endianess of the target processor and will give different results on big- and little-endian CPUs.

 © 2019-2024 SEGGER Microcontroller GmbH

4.2 Using the volatile keyword

Using the volatile keyword when declaring a variable ensures that the compiler does not optimize any use of the variable on the assumption that this variable is unused or unmodified. The declaration of a variable as volatile tells the compiler that the variable can be modified at any time externally to the implementation, for example:

- By another thread or an interrupt routine or signal handler.
- By hardware.

A volatile declaration prevents the compiler from:

- Eliminating a read access to the variable because it's value is already known or held in a CPU register.
- Eliminating a write access to the variable because it's never used afterwards.
- Combining the memory access to the variable with other read/write accesses, see *Unaligned memory access*.
- Changing the type of the variable.
- Completely eliminating the variable.

In practice, you must declare a variable as volatile when:

- Accessing memory-mapped peripherals.
- Sharing global variables between multiple threads.
- Accessing global variables that may be modified in an interrupt routine or signal handler.
- Implementing time delays using a counting loop.
- Accessing memory that might be modified via DMA.

If you do not use the volatile keyword where it is needed, then the compiler might optimize accesses to the variable and generate unintended code or remove intended functionality.

Example 1

```
uint32_t State;

int func() {
  State = 0;
  ...
  // Interrupt may occur changing 'State'
  ...
  if (State != 0) {
     ...
  }
}
```

The compiler may remove the complete if-clause as it didn't expect that `State` is modified outside the function.

Example 2

```
uint32_t *pStatus = (uint32_t)0xC0010300; // The status register of a peripheral
while (*pStatus & BUSY_BIT) {}            // Wait until the peripheral gets ready
```

This may result in an endless loop as the status register may be read only once before the loop.

Example 3

```
for (int Count = 0; Count < 100000; ++Count) {}     // Time delay
```

The compiler may remove the whole loop, because the value of `Count` is never used.

Example 4

```
static uint32_t State;
```

 © 2019-2024 SEGGER Microcontroller GmbH

```
void funcA() {
  ...
  if (...) State = 0;
  else     State = 4711;
  ...
}

int funcB() {
  if (State == 4711) return 0;
  ...
}
```

The compiler may change the type of `State` into `bool`, because `State` can only take two different values.

4.3 Function inlining

Function inlining means that a function, whose definition is known at compile time, is integrated into the body of its caller to eliminate the overhead of the function call. This optimization is used to increase execution speed and/or reduce code size. The compiler decides which functions are actually inlined using internal heuristics considering the selected optimization goal. Function inlining may significantly increase code size if optimizing for speed.

4.3.1 Controlling inlining

Function inlining can be controlled via keywords or functions attributes in the source code and via compiler command line options.

Change inline behavior of individual functions:

Function keyword or attribute	Description
`inline`	Specifying this keyword on a function definition or declaration gives the compiler a hint to favor inlining of the function. However, it's still the decision of the compiler whether to inline the function or not.
`__attribute__((always_inline))`	Specifying this function attribute on a function definition or declaration tells the compiler to always inline the function if possible.
`__attribute__((noinline))`	Specifying this function attribute on a function definition or declaration tells the compiler to never inline the function.

Globally change inline behavior:

Command line option	Description
`-fno_inline`	Disable inlining except for functions with the **`always_inline`** attribute
`-finline-hint-functions`	Disable inlining of function that have neither the **`always_inline`** attribute nor the **`inline`** keyword specified

 © 2019-2024 SEGGER Microcontroller GmbH

4.4 Undefined behavior

The C and C++ standards consider any code that uses non-portable, erroneous program or data constructs as undefined behavior. SEGGER provides no information or guarantees about the behavior of the compiler when presented with a program that exhibits undefined behavior. That includes whether the compiler attempts to diagnose the undefined behavior.

 © 2019-2024 SEGGER Microcontroller GmbH

Chapter 5

Link Time Optimization

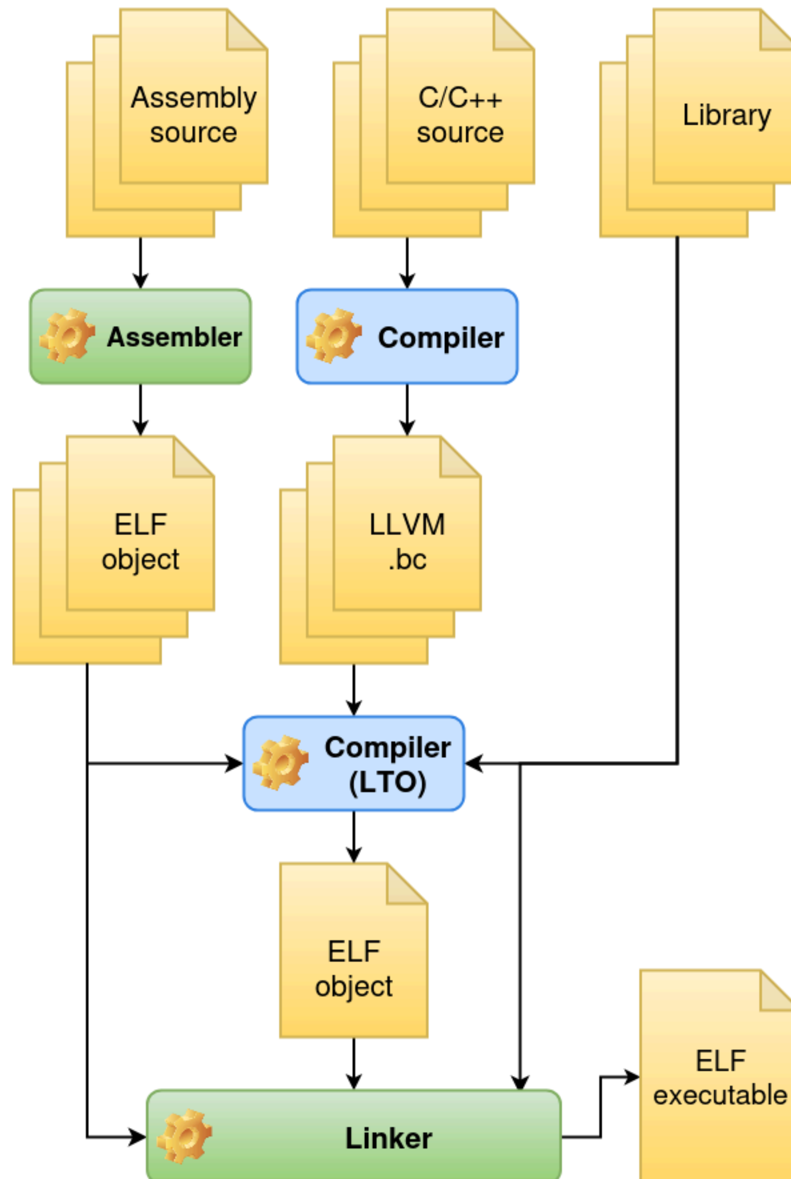
Link Time Optimization (LTO) is another name for intermodular optimization performed on the whole application code. Much more efficient code can be generated when the compiler handles the complete application code instead of optimizing each single C/C++ module separately.

This chapter shows how to use the compiler for LTO.

© 2019-2024 SEGGER Microcontroller GmbH

5.1 Typical build process with LTO

The name "*Link* Time Optimization" may be misleading, because LTO must not be confused with the operation performed by the linker. The picture shows a typical build process when using LTO.



Like in a normal build process each C/C++ source file must be translated by a separate compiler run, but for LTO the option **`-emit-llvm-bc`** must be used in order to create a LLVM language output file (with extension `.bc`) for each source file.

In a second (LTO-) step, the compiler processes all LLVM language files in a single run, creating a single ELF object file as result. All other ELF object files (and libraries) that should be linked into the final application must be provided to the compiler, too. These files are not translated or merged into the output file, instead they are only used to scan for symbols to be preserved in the output.

Example

In a project there is usually some startup code written in assembly language, that calls the `main()` function: A `Startup.o` object file generated from a `Startup.asm` file by an assembler. The function `main()` itself is located in a C/C++ file that is processed by the

 © 2019-2024 SEGGER Microcontroller GmbH

compiler during a LTO run. If the compiler can't see the `Startup.o` file, it may consider the function `main()` to be never called and remove it during optimization.

After the LTO pass the resulting object file has to be linked with the other ELF object files and libraries using a linker tool in order to get an executable application.

While using LTO usually generates better target code, there is one disadvantage: The compile time for partially rebuilds. If only a single C source file has changed a rebuild of the application without LTO only requires to compile the single source file and perform the link step. With using LTO the optimization and code generation for all C/C++ modules have to be processed, which usually takes more time.

 © 2019-2024 SEGGER Microcontroller GmbH

5.2 Compiler invocation for LTO

Command line syntax to execute a LTO pass:

```
segger-cc -cc1lto <options> <output-type> <input-files>... -o <output-file>
```

Exactly the same options can be used as for a normal compiler run with the following differences:

1) Multiple input files can be specified, each must be either:

- A LLVM language (.bc) file
- A relocatable ELF object file
- A library containing ELF object files

2) `<output-type>` must be one of **-c** / **-emit-obj** or **-S**.

3) Additional options are available:

Option	Description
-input-list `<file>`	The compiler reads a list of input files from `<file>` which must be a text file containing one file name or file path per line.
-output-list `<file>`	The compiler writes a list of object files into `<file>`
--keep-symbol `<symbol>`	Requests that the compiler keeps `<symbol>` in the generated output that may otherwise be discarded by optimization.

Mandatory options for LTO

- *Specifying the target architecture*
- *Specifying target features*
- *Specifying an optimization level*

Even if the same options can be used for both the compile and LTO invocation of the compiler, some option are only handled by one of these passes. For example: Language options like **-W**, **-D** or **-std=** can be specified, but are silently ignored by the LTO pass.

 © 2019-2024 SEGGER Microcontroller GmbH

Chapter 6

Compiler options reference

© 2019-2024 SEGGER Microcontroller GmbH

6.1 Undocumented compiler features

The SEGGER Compiler is built on LLVM technology and preserves the functionality of that technology where possible. This means that there are additional features available in the compiler that are not listed in this documentation. For information on these features, see the The Clang Compiler User's Manual.

Any features not documented in this manual are not supported and are used at your own risk. You are responsible for making sure that any generated code using these features is operating correctly.

 © 2019-2024 SEGGER Microcontroller GmbH

6.2 List of all compiler options

Options marked with a '*' in the last column can be used for the invocation of the integrated assembler, see *Running the compiler on assembly files*.

Option	Description	
-c	Only run preprocess, compile, and assemble steps	*
-D <macro>=<value>	Define <macro> to <value> (or 1 if <value> omitted)	
-debug-info-kind=standalone	Generate complete debug info	*
-debug-info-macro	Emit macro debug information	*
-debugger-tuning=gdb	Tune debug info for gdb	*
-dependency-file <value>	Filename (or -) to write dependency output to	
-dI	Print include directives in -E mode in addition to normal output	
-disable-free	Disable freeing of memory on exit	
-disable-llvm-verifier	Don't run the LLVM IR verifier pass	
-dM	Print macro definitions in -E mode instead of normal output	
-dwarf-version=<value>	Set DWARF version to <value>	*
-E	Only run the preprocessor	
-emit-llvm-bc	Build ASTs then convert to LLVM, emit .bc file	
-emit-obj	Emit native object files	
-exception-model=dwarf	Set exception model to “DWARF”	
-fansi-escape-codes	Use ANSI escape codes for diagnostics	
-fcolor-diagnostics	Enable colors in diagnostics	
-fcommon	Place uninitialized global variables in a common block	
-fcxx-exceptions	Enable C++ exceptions	
-fdata-sections	Place each data in its own section (Default)	
-ffunction-sections	Place each function in its own section (Default)	
-fgnuc-version=<value>	Sets various macros to claim compatibility with the given GCC version (default is 4.2.1)	
-finline-hint-functions	Inline functions which are (explicitly or implicitly) marked inline	
-finstrument-functions	Generate calls to instrument function entry and exit	
-fmath-errno	Require math functions to indicate errors by setting errno	
-fnative-half-arguments-and-returns	Use the native __fp16 type for arguments and returns (and skip ABI-specific lowering)	
-fnative-half-type	Use the native half type for __fp16 instead of promoting to float	
-fno-builtin	Disable implicit builtin knowledge of functions	
-fno-caret-diagnostics	Disable caret line in diagnostic messages	
-fno-common	Compile common globals like normal definitions	
-fno-data-sections	Use one data section for each source file	
-fno-diagnostics-fixit-info	Do not include fixit information in diagnostics	

 © 2019-2024 SEGGER Microcontroller GmbH

Option	Description	
-fno-function-sections	Use one text section for each source file	
-fno-inline	Prevent inlining of functions	
-fno-rtti	Disable generation of rtti information	
-fno-signed-char	char is unsigned	
-fshort-enums	Allocate to an enum type only as many bytes as it needs for the declared range of possible values	
-fstack-size-section	Emit section containing metadata on function stack sizes	
-funwind-tables=0	Don't generate unwinding tables	
-funwind-tables=1	Generate "synchronous" unwinding tables for all functions	
-funwind-tables=2	Generate "asynchronous" unwinding tables (instr precise) for all functions	
-fwchar-type=<value>	Select underlying type for `wchar_t`, <value> is one of "char", "short" or "int"	
-gpubnames	Generate Dwarf `.debug_pubnames` and `.debug_pubtypes` sections	
-help	Display available options	*
-I <dir>	Add directory to the end of the list of include search paths	
-imacros <file>	Include macros from file before parsing	
-include <file>	Include file before parsing	
-iquote <directory>	Add directory to QUOTE include search path	
-isystem <directory>	Add directory to SYSTEM include search path	
-main-file-name <value>	Main file name to use for debug info and source if missing	
-marm	Use ARM instruction set (ARM only)	
-mbig-endian	Target has little big byte order set (ARM only)	*
-mbss=<value>	name the .bss section	
-mconstructor-aliases	Enable emitting complete constructors and destructors as aliases when possible	
-mdata=<value>	name the .data section	
-mfloat-abi hard	Generate VFP code to implement floating point operations and use VFP registers to pass floating point parameters on function calls	
-mfloat-abi soft	Don't use any floating point unit, generate calls to the C library to implement floating point operations	
-mfloat-abi softfp	Generate VFP code to implement floating point operations but use general purpose registers to pass floating point parameters on function calls	
-mframe-pointer=all	Keep all frame pointers	
-mframe-pointer=nonleaf	Keep frame pointers in non-leaf functions	
-mhard-float	Generate VFP code to implement floating point operations and use VFP registers to pass floating point parameters on function calls	
-mlittle-endian	Target has little endian byte order set (ARM only)	*

© 2019-2024 SEGGER Microcontroller GmbH

Option	Description	
-mllvm -arm-add-build-attributes	Emit the build attributes that depend on the hardware into the object file	*
-mllvm -arm-global-merge=false	Don't merge globals	
-mllvm -arm-global-merge=true	Merges globals with internal linkage into one, so that globals which were merged can be addressed using offsets from the same base pointer	
-mllvm -generate-arange-section	Generate DWARF aranges in the output file	
-mlong-double-128	Force long double to be 128 bits	
-mno-thumb	Use ARM instruction set (ARM only)	*
-mrodata=<value>	name the .rodata section	
-msoft-float	Don't use any floating point unit, generate calls to the C library to implement floating point operations	
-mstack-overflow-check	Generate stub for stack checking in every function	
-MT <value>	Specify name of main file output in depfile	
-mtext=<value>	name the .text section	
-mthread-model posix	Use POSIX thread model	
-mthread-model single	Compile for a Single Threaded Environment	
-mthumb	Use THUMB instruction set (ARM only)	*
-nobuiltininc	Disable builtin #include directories	
-nostdsysteminc	Disable standard system #include directories	
-o <file>	Write output to <file>	
-O<level>	Use optimization <level>	
-S	Only run preprocess and compilation steps	
-std=<value>	Language standard to compile for	
-sys-header-deps	Include system headers in dependency output	
-target-abi aapcs	Use the Procedure Call Standard for the ARM Architecture	
-target-abi ilp32	Use 32-bit integer ABI for RISC-V	
-target-abi ilp32d	Use 32-bit integer ABI for RISC-V with double precision floating point	
-target-abi ilp32e	Use 32-bit integer ABI for RISC-V with 16 registers	
-target-abi ilp32f	Use 32-bit integer ABI for RISC-V with single precision floating point	
-target-cpu <value>	Target a specific cpu type	
-target-feature <value>	Target specific attributes	*
--target=<value>	Generate code for the given target	*
-triple <value>	Specify target triple (e.g. i686-apple-darwin9)	*
-U <macro>	Undefine macro <macro>	
-version	Print the compiler version	
-W<group>	Enable warnings of diagnostic group <group>	

© 2019-2024 SEGGER Microcontroller GmbH

Option	Description	
-Wall	This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid	
-Wdeprecated	Enable warnings for deprecated constructs and define `__DEPRECATED`	
-Werror	Turn all warnings into errors	
-Werror=<group>	Turn all messages of diagnostic group <group> into errors	
-Wextra	This enables some extra warning messages that are not enabled by -Wall	
-Wno-<group>	Disable warnings of diagnostic group <group>	
-Wno-error=<group>	Turn all messages of diagnostic group <group> into warnings	
-Wpedantic	Issue warning when violations of the selected language standard are detected	
-Wpedantic-errors	Issue errors when violations of the selected language standard are detected	
-x assembler-with-cpp	Treat input file as assembly type file. Can be used only to run the preprocessor on assembly files together with option -E	
-x c	Treat input file as having type C	
-x c++	Treat input file as having type C++	

 © 2019-2024 SEGGER Microcontroller GmbH

6.3 Predefined macros

The compiler predefines various macros which provide information about toolchain version numbers and compiler options. The complete list of predefined macros is shown when adding the options **`-E -dM`** at compiler invocation.

Macro name	Value	Description
`__STDC_VERSION__`	Month	Defined when compiling C code, and set to a value that identifies the targeted C standard. The value is the month of the release of the standard (numeric decimal YYYYMM).
`__cplusplus`	Month	Defined when compiling C++ code, and set to a value that identifies the targeted C++ standard. The value is the month of the release of the standard (numeric decimal YYYYMM).
`__SEGGER_CC__`	Numeric	The major version number of the compiler
`__SEGGER_CC_MINOR__`	Numeric	The minor version number of the compiler
`__SEGGER_CC_PATCHLEVEL__`	Numeric	The patch level of the compiler
`__SEGGER_CC_VERSION__`	String	The full version of the compiler as string (“major.minor.patch”)
`__VERSION__`	String	Compiler name and version

`__GNUC__`
`__GNUC_MINOR__`
`__GNUC_PATCHLEVEL__`
	Numeric	Can be set with the compiler option **`-fgnuc-version`** to claim gcc compatibility

`__BIG_ENDIAN__`	1	Set if compiling for a big endian target
`__LITTLE_ENDIAN__`	1	Set if compiling for a big little target

6.3.1 Predefined macros for ARM targets

Macro name	Value	Description
`__arm__`	1	Set if compiling for ARM targets.
`__ARM_ARCH`	Numeric	Specifies the version of the target architecture
`__ARM_ARCH_`**`<arch>`**`__`	1	Set if compiling for the corresponding target architecture, in which **`<arch>`** is one of: `6M 6 7A 7EM 7M 7R 7S 8A 8M_BASE 8M_MAIN 8R 8_1A 8_1M_MAIN 8_2A 8_3A 8_4A 8_5A 8_6A`
`__ARM_ARCH_PROFILE`	Character	Specifies the profile of the target architecture: 'A', 'R' or 'M'.
`__ARM_FEATURE_CLZ`	1	Set if the CLZ (count leading zeroes) instruction is supported in hardware.
`__ARM_FP`	Numeric	Set if hardware floating-point is available. Bits 1-3 indicate the supported floating-point precision levels: Bit 1 - half precision (16-bit). Bit 2 - single precision (32-bit). Bit 3 - double precision (64-bit).
`__ARM_ARCH_ISA_ARM`	1	Set if the compiler is creating ARM instructions.
`__ARM_ARCH_ISA_THUMB`	Numeric	Set if the compiler is creating Thumb instructions. Value is 1 for Thumb-1, 2 for Thumb-2.

 © 2019-2024 SEGGER Microcontroller GmbH

6.3.2 Predefined macros for RISC-V targets

Macro name	Value	Description
__riscv	1	Set if compiling for RISC-V targets
__riscv_**<ext>**	Version	Set if the target has extension **<ext>**. Value is MajorVersion * 1000000 + MinorVersion * 1000.

 © 2019-2024 SEGGER Microcontroller GmbH

6.4 Complete list of supported target features

Target feature	Description
ARM targets	
8msecext	Enable support for ARMv8-M Security Extensions.
a12	Cortex-A12 ARM processors.
a15	Cortex-A15 ARM processors.
a17	Cortex-A17 ARM processors.
a35	Cortex-A35 ARM processors.
a5	Cortex-A5 ARM processors.
a53	Cortex-A53 ARM processors.
a55	Cortex-A55 ARM processors.
a57	Cortex-A57 ARM processors.
a7	Cortex-A7 ARM processors.
a72	Cortex-A72 ARM processors.
a73	Cortex-A73 ARM processors.
a75	Cortex-A75 ARM processors.
a76	Cortex-A76 ARM processors.
a77	Cortex-A77 ARM processors.
a78c	Cortex-A78C ARM processors.
a8	Cortex-A8 ARM processors.
a9	Cortex-A9 ARM processors.
aclass	Is application profile ('A' series).
acquire-release	Has v8 acquire/release (lda/ldaex etc) instructions.
aes	Enable AES support.
armv4	ARMv4 architecture.
armv4t	ARMv4t architecture.
armv5te	ARMv5te architecture.
armv5tej	ARMv5tej architecture.
armv6	ARMv6 architecture.
armv6-m	ARMv6m architecture.
armv6kz	ARMv6kz architecture.
armv6t2	ARMv6t2 architecture.
armv7-a	ARMv7a architecture.
armv7-m	ARMv7m architecture.
armv7-r	ARMv7r architecture.
armv7e-m	ARMv7em architecture.
armv8-a	ARMv8a architecture.
armv8-m.base	ARMv8mBaseline architecture.
armv8-m.main	ARMv8mMainline architecture.
armv8-r	ARMv8r architecture.
armv8.1-m.main	ARMv81mMainline architecture.
armv8.2-a	ARMv82a architecture.
armv9-a	ARMv9a architecture.

 © 2019-2024 SEGGER Microcontroller GmbH

Target feature	Description
avoid-partial-cpsr	Avoid CPSR partial update for OOO execution.
bf16	Enable support for BFloat16 instructions.
cheap-predica-ble-cpsr	Disable +1 predication cost for instructions updating CPSR.
cortex-a710	Cortex-A710 ARM processors.
cortex-a78	Cortex-A78 ARM processors.
cortex-x1	Cortex-X1 ARM processors.
cortex-x1c	Cortex-X1C ARM processors.
crc	Enable support for CRC instructions.
crypto	Enable support for Cryptography extensions.
d32	Extend FP to 32 double registers.
db	Has data barrier (dmb/dsb) instructions.
dfb	Has full data barrier (dfb) instruction.
dont-widen-vmovs	Don't widen VMOVS to VMOVD.
dotprod	Enable support for dot product instructions.
dsp	Supports DSP instructions in ARM and/or Thumb2.
expand-fp-mlx	Expand VFP/NEON MLA/MLS instructions.
fix-cmse-cve-2021-35465	Mitigate against the cve-2021-35465 security vulnurability.
fix-cortex-a57-aes-1742098	Work around Cortex-A57 Erratum 1742098 / Cortex-A72 Erratum 1655431 (AES).
fp-armv8	Enable ARMv8 FP.
fp-armv8d16	Enable ARMv8 FP with only 16 d-registers.
fp-armv8d16sp	Enable ARMv8 FP with only 16 d-registers and no double precision.
fp-armv8sp	Enable ARMv8 FP with no double precision.
fp16	Enable half-precision floating point.
fp16fml	Enable full half-precision floating point fml instructions.
fp64	Floating point unit supports double precision.
fpao	Enable fast computation of positive address offsets.
fpregs	Enable FP registers.
fpregs16	Enable 16-bit FP registers.
fpregs64	Enable 64-bit FP registers.
fullfp16	Enable full half-precision floating point.
hwdiv	Enable divide instructions in Thumb.
hwdiv-arm	Enable divide instructions in ARM mode.
i8mm	Enable Matrix Multiply Int8 Extension.
lob	Enable Low Overhead Branch extensions.
long-calls	Generate calls via indirect call instructions.
loop-align	Prefer 32-bit alignment for loops.
m3	Cortex-M3 ARM processors.
m7	Cortex-M7 ARM processors.
mclass	Is microcontroller profile ('M' series).
mp	Supports Multiprocessing extension.

Target feature	Description
muxed-units	Has muxed AGU and NEON/FPU.
mve	Support M-Class Vector Extension with integer ops.
mve.fp	Support M-Class Vector Extension with integer and floating ops.
mve1beat	Model MVE instructions as a 1 beat per tick architecture.
neon	Enable NEON instructions.
neon-fpmovs	Convert VMOVSR, VMOVRS, VMOVS to NEON.
no-branch-predictor	Has no branch predictor.
no-movt	Don't use movt/movw pairs for 32-bit imms.
noarm	Does not support ARM mode execution.
nonpipelined-vfp	VFP instructions are not pipelined.
pacbti	Enable Pointer Authentication and Branch Target Identification.
perfmon	Enable support for Performance Monitor extensions.
prefer-vmovsr	Prefer VMOVSR.
r4	Cortex-R4 ARM processors.
r5	Cortex-R5 ARM processors.
r52	Cortex-R52 ARM processors.
r7	Cortex-R7 ARM processors.
ras	Enable Reliability, Availability and Serviceability extensions.
rclass	Is realtime profile ('R' series).
ret-addr-stack	Has return address stack.
sb	Enable v8.5a Speculation Barrier.
sha2	Enable SHA1 and SHA256 support.
slow-fp-brcc	FP compare + branch is slow.
slowfpvfmx	Disable VFP / NEON FMA instructions.
slowfpvmlx	Disable VFP / NEON MAC instructions.
soft-float	Use software floating point features..
splat-vfp-neon	Splat register from VFP to NEON.
strict-align	Disallow all unaligned memory access.
thumb-mode	Thumb mode.
thumb2	Enable Thumb2 instructions.
trustzone	Enable support for TrustZone security extensions.
use-mipipeliner	Use the MachinePipeliner.
use-misched	Use the MachineScheduler.
v4t	Support ARM v4T instructions.
v5t	Support ARM v5T instructions.
v5te	Support ARM v5TE, v5TEj, and v5TExp instructions.
v6	Support ARM v6 instructions.
v6k	Support ARM v6k instructions.
v6m	Support ARM v6M instructions.
v6t2	Support ARM v6t2 instructions.
v7	Support ARM v7 instructions.
v7clrex	Has v7 clrex instruction.
v8	Support ARM v8 instructions.

 © 2019-2024 SEGGER Microcontroller GmbH

Target feature	Description
v8.1a	Support ARM v8.1a instructions.
v8.1m.main	Support ARM v8-1M Mainline instructions.
v8.2a	Support ARM v8.2a instructions.
v8.3a	Support ARM v8.3a instructions.
v8.4a	Support ARM v8.4a instructions.
v8.5a	Support ARM v8.5a instructions.
v8m	Support ARM v8M Baseline instructions.
v8m.main	Support ARM v8M Mainline instructions.
v9a	Support ARM v9a instructions.
vfp2	Enable VFP2 instructions.
vfp2sp	Enable VFP2 instructions with no double precision.
vfp3	Enable VFP3 instructions.
vfp3d16	Enable VFP3 instructions with only 16 d-registers.
vfp3d16sp	Enable VFP3 instructions with only 16 d-registers and no double precision.
vfp3sp	Enable VFP3 instructions with no double precision.
vfp4	Enable VFP4 instructions.
vfp4d16	Enable VFP4 instructions with only 16 d-registers.
vfp4d16sp	Enable VFP4 instructions with only 16 d-registers and no double precision.
vfp4sp	Enable VFP4 instructions with no double precision.
virtualization	Supports Virtualization extension.
vldn-align	Check for VLDn unaligned access.
vmlx-forwarding	Has multiplier accumulator forwarding.
vmlx-hazards	Has VMLx hazards.
RISCV targets	
a	'A' (Atomic Instructions).
c	'C' (Compressed Instructions).
d	'D' (Double-Precision Floating-Point).
e	Implements RV{32,64}E (provides 16 rather than 32 GPRs).
f	'F' (Single-Precision Floating-Point).
fast-unaligned-access	Has reasonably performant unaligned loads and stores (both scalar and vector).
m	'M' (Integer Multiplication and Division).
relax	Enable Linker relaxation..
v	'V' (Vector Extension for Application Processors).
zba	'Zba' (Address Generation Instructions).
zbb	'Zbb' (Basic Bit-Manipulation).
zbc	'Zbc' (Carry-Less Multiplication).
zbkb	'Zbkb' (Bitmanip instructions for Cryptography).
zbkx	'Zbkx' (Crossbar permutation instructions).
zbs	'Zbs' (Single-Bit Instructions).
zmmul	'Zmmul' (Integer Multiplication).

 © 2019-2024 SEGGER Microcontroller GmbH

6.5 Implicit applied target features for ARM CPUs

CPU	Implicit target features
arm-arm1136j-s	+armv6 +dsp +v4t +v5t +v5te +v6
arm-arm1136jf-s	+armv6 +dsp +fp64 +fpregs +fpregs64 +slowfpvmlx +v4t +v5t +v5te +v6 +vfp2 +vfp2sp
arm-arm1156t2-s	+armv6t2 +dsp +thumb2 +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v8m
arm-arm1156t2f-s	+armv6t2 +dsp +fp64 +fpregs +fpregs64 +slowfpvmlx +thumb2 +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v8m +vfp2 +vfp2sp
arm-arm1176jz-s	+armv6kz +dsp +trustzone +v4t +v5t +v5te +v6 +v6k
arm-arm1176jzf-s	+armv6kz +dsp +fp64 +fpregs +fpregs64 +slowfpvmlx +trustzone +v4t +v5t +v5te +v6 +v6k +vfp2 +vfp2sp
arm-arm710t	+armv4t +v4t
arm-arm720t	+armv4t +v4t
arm-arm7tdmi	+armv4t +v4t
arm-arm7tdmi-s	+armv4t +v4t
arm-arm8	+armv4
arm-arm810	+armv4
arm-arm9	+armv4t +v4t
arm-arm920	+armv4t +v4t
arm-arm920t	+armv4t +v4t
arm-arm922t	+armv4t +v4t
arm-arm926ej-s	+armv5te +armv5tej +dsp +v4t +v5t +v5te
arm-arm940t	+armv4t +v4t
arm-arm946e-s	+armv5te +dsp +v4t +v5t +v5te
arm-arm966e-s	+armv5te +dsp +v4t +v5t +v5te
arm-arm968e-s	+armv5te +dsp +v4t +v5t +v5te
arm-arm9e	+armv5te +dsp +v4t +v5t +v5te
arm-arm9tdmi	+armv4t +v4t
arm-cortex-a12	+a12 +aclass +armv7-a +avoid-partial-cpsr +d32 +db +dsp +fp16 +fp64 +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +neon +perfmon +ret-addr-stack +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization +vmlx-forwarding
arm-cortex-a15	+a15 +aclass +armv7-a +avoid-partial-cpsr +d32 +db +dont-widen-vmovs +dsp +fp16 +fp64 +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +muxed-units +neon +perfmon +ret-addr-stack +splat-vfp-neon +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization +vldn-align
arm-cortex-a17	+a17 +aclass +armv7-a +avoid-partial-cpsr +d32 +db +dsp +fp16 +fp64 +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +neon +perfmon +ret-addr-stack +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16

 © 2019-2024 SEGGER Microcontroller GmbH

CPU	Implicit target features
	+vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization +vmlx-forwarding
arm-cortex-a32	+aclass +acquire-release +aes +armv8-a +crc +crypto +d32 +db +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +neon +perfmon +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a35	+a35 +aclass +acquire-release +aes +armv8-a +crc +crypto +d32 +db +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +neon +perfmon +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a5	+a5 +aclass +armv7-a +d32 +db +dsp +fp16 +fp64 +fpregs +fpregs64 +mp +neon +perfmon +ret-addr-stack +slow-fp-brcc +slowfpvfmx +slowfpvmlx +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +vmlx-forwarding
arm-cortex-a53	+a53 +aclass +acquire-release +aes +armv8-a +crc +crypto +d32 +db +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpao +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +neon +perfmon +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a55	+a55 +aclass +acquire-release +aes +armv8.2-a +crc +crypto +d32 +db +dotprod +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +hwdiv-arm +mp +neon +perfmon +ras +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8.1a +v8.2a +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a57	+a57 +aclass +acquire-release +aes +armv8-a +avoid-partial-cpsr +cheap-predicable-cpsr +crc +crypto +d32 +db +dsp +fix-cortex-a57-aes-1742098 +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpao +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +neon +perfmon +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a7	+a7 +aclass +armv7-a +d32 +db +dsp +fp16 +fp64 +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +neon +perfmon +ret-addr-stack +slow-fp-brcc +slowfpvfmx +slowfpvmlx +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp

 © 2019-2024 SEGGER Microcontroller GmbH

CPU	Implicit target features
	+vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization +vmlx-forwarding +vmlx-hazards
arm-cortex-a710	+aclass +acquire-release +armv9-a +bf16 +cortex-a710 +crc +d32 +db +dotprod +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp16fml +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +hwdiv-arm +i8mm +mp +neon +perfmon +ras +sb +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8.1a +v8.2a +v8.3a +v8.4a +v8.5a +v8m +v9a +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a72	+a72 +aclass +acquire-release +aes +armv8-a +crc +crypto +d32 +db +dsp +fix-cortex-a57-aes-1742098 +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +neon +perfmon +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a73	+a73 +aclass +acquire-release +aes +armv8-a +crc +crypto +d32 +db +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +neon +perfmon +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a75	+a75 +aclass +acquire-release +aes +armv8.2-a +crc +crypto +d32 +db +dotprod +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +hwdiv-arm +mp +neon +perfmon +ras +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8.1a +v8.2a +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a76	+a76 +aclass +acquire-release +aes +armv8.2-a +crc +crypto +d32 +db +dotprod +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +hwdiv-arm +mp +neon +perfmon +ras +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8.1a +v8.2a +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a76ae	+a76 +aclass +acquire-release +aes +armv8.2-a +crc +crypto +d32 +db +dotprod +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +hwdiv-arm +mp +neon +perfmon +ras +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8.1a +v8.2a +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a77	+a77 +aclass +acquire-release +aes +armv8.2-a +crc +crypto +d32 +db +dotprod +dsp +fp-armv8

 © 2019-2024 SEGGER Microcontroller GmbH

CPU	Implicit target features
	+fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +hwdiv-arm +mp +neon +perfmon +ras +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8.1a +v8.2a +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a78	+aclass +acquire-release +aes +armv8.2-a +cortex-a78 +crc +crypto +d32 +db +dotprod +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +hwdiv-arm +mp +neon +perfmon +ras +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8.1a +v8.2a +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a78c	+a78c +aclass +acquire-release +aes +armv8.2-a +crc +crypto +d32 +db +dotprod +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +hwdiv-arm +mp +neon +perfmon +ras +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8.1a +v8.2a +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-a8	+a8 +aclass +armv7-a +d32 +db +dsp +fp64 +fpregs +fpregs64 +neon +nonpipelined-vfp +perfmon +ret-addr-stack +slow-fp-brcc +slowfpvfmx +slowfpvmlx +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vmlx-forwarding +vmlx-hazards
arm-cortex-a9	+a9 +aclass +armv7-a +avoid-partial-cpsr +d32 +db +dsp +expand-fp-mlx +fp16 +fp64 +fpregs +fpregs64 +mp +muxed-units +neon +neon-fpmovs +perfmon +prefer-vmovsr +ret-addr-stack +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vldn-align +vmlx-forwarding +vmlx-hazards
arm-cortex-m0	+armv6-m +db +mclass +no-branch-predictor +noarm +strict-align +thumb-mode +v4t +v5t +v5te +v6 +v6m
arm-cortex-m0plus	+armv6-m +db +mclass +no-branch-predictor +noarm +strict-align +thumb-mode +v4t +v5t +v5te +v6 +v6m
arm-cortex-m1	+armv6-m +db +mclass +no-branch-predictor +noarm +strict-align +thumb-mode +v4t +v5t +v5te +v6 +v6m
arm-cortex-m23	+8msecext +acquire-release +armv8-m.base +db +hwdiv +mclass +no-branch-predictor +no-movt +noarm +strict-align +thumb-mode +v4t +v5t +v5te +v6 +v6m +v7clrex +v8m
arm-cortex-m3	+armv7-m +db +hwdiv +loop-align +m3 +mclass +no-branch-predictor +noarm +thumb-mode +thumb2 +use-misched +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m
arm-cortex-m33	+8msecext +acquire-release +armv8-m.main +db +dsp +fix-cmse-cve-2021-35465 +fp-armv8d16sp +fp16 +fpregs +hwdiv +loop-align +mclass +no-branch-predictor

 © 2019-2024 SEGGER Microcontroller GmbH

CPU	Implicit target features
	+noarm +slowfpvfmx +slowfpvmlx +thumb-mode +thumb2 +use-misched +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +v8m.main +vfp2sp +vfp3d16sp +vfp4d16sp
arm-cortex-m35p	+8msecext +acquire-release +armv8-m.main +db +dsp +fix-cmse-cve-2021-35465 +fp-armv8d16sp +fp16 +fpregs +hwdiv +loop-align +mclass +no-branch-predictor +noarm +slowfpvfmx +slowfpvmlx +thumb-mode +thumb2 +use-misched +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +v8m.main +vfp2sp +vfp3d16sp +vfp4d16sp
arm-cortex-m4	+armv7e-m +db +dsp +fp16 +fpregs +hwdiv +loop-align +mclass +no-branch-predictor +noarm +slowfpvfmx +slowfpvmlx +thumb-mode +thumb2 +use-misched +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2sp +vfp3d16sp +vfp4d16sp
arm-cortex-m52	+8msecext +acquire-release +armv8.1-m.main +db +dsp +fp-armv8d16 +fp-armv8d16sp +fp16 +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +lob +loop-align +mclass +mve +mve.fp +mve1beat +no-branch-predictor +noarm +pacbti +ras +slowfpvmlx +thumb-mode +thumb2 +use-misched +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8.1m.main +v8m +v8m.main +vfp2 +vfp2sp +vfp3d16 +vfp3d16sp +vfp4d16 +vfp4d16sp
arm-cortex-m55	+8msecext +acquire-release +armv8.1-m.main +db +dsp +fix-cmse-cve-2021-35465 +fp-armv8d16 +fp-armv8d16sp +fp16 +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +lob +loop-align +mclass +mve +mve.fp +no-branch-predictor +noarm +ras +slowfpvmlx +thumb-mode +thumb2 +use-misched +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8.1m.main +v8m +v8m.main +vfp2 +vfp2sp +vfp3d16 +vfp3d16sp +vfp4d16 +vfp4d16sp
arm-cortex-m7	+armv7e-m +db +dsp +fp-armv8d16 +fp-armv8d16sp +fp16 +fp64 +fpregs +fpregs64 +hwdiv +m7 +mclass +noarm +thumb-mode +thumb2 +use-mipipeliner +use-misched +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2 +vfp2sp +vfp3d16 +vfp3d16sp +vfp4d16 +vfp4d16sp
arm-cortex-m85	+8msecext +acquire-release +armv8.1-m.main +db +dsp +fp-armv8d16 +fp-armv8d16sp +fp16 +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +lob +mclass +mve +mve.fp +noarm +pacbti +ras +thumb-mode +thumb2 +use-misched +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8.1m.main +v8m +v8m.main +vfp2 +vfp2sp +vfp3d16 +vfp3d16sp +vfp4d16 +vfp4d16sp
arm-cortex-r4	+armv7-r +avoid-partial-cpsr +db +dsp +hwdiv +perfmon +r4 +rclass +ret-addr-stack +thumb-mode +thumb2 +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m
arm-cortex-r4f	+armv7-r +avoid-partial-cpsr +db +dsp +fp64 +fpregs +fpregs64 +hwdiv +perfmon +r4 +rclass +ret-addr-stack +slow-fp-brcc +slowfpvfmx +slowfpvmlx +thumb-mode +thumb2 +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2 +vfp2sp +vfp3d16 +vfp3d16sp
arm-cortex-r5	+armv7-r +avoid-partial-cpsr +db +dsp +fp64 +fpregs +fpregs64 +hwdiv +hwdiv-arm +perfmon +r5 +rclass

 © 2019-2024 SEGGER Microcontroller GmbH

CPU	Implicit target features
	+ret-addr-stack +slow-fp-brcc +slowfpvfmx +slowfpvmlx +thumb-mode +thumb2 +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2 +vfp2sp +vfp3d16 +vfp3d16sp
arm-cortex-r52	+acquire-release +armv8-r +crc +d32 +db +dfb +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpao +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +neon +perfmon +r52 +rclass +thumb-mode +thumb2 +use-misched +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-r7	+armv7-r +avoid-partial-cpsr +db +dsp +fp16 +fp64 +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +perfmon +r7 +rclass +ret-addr-stack +slow-fp-brcc +slowfpvfmx +slowfpvmlx +thumb-mode +thumb2 +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2 +vfp2sp +vfp3d16 +vfp3d16sp
arm-cortex-r8	+armv7-r +avoid-partial-cpsr +db +dsp +fp16 +fp64 +fpregs +fpregs64 +hwdiv +hwdiv-arm +mp +perfmon +rclass +ret-addr-stack +slow-fp-brcc +slowfpvfmx +slowfpvmlx +thumb-mode +thumb2 +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8m +vfp2 +vfp2sp +vfp3d16 +vfp3d16sp
arm-cortex-x1	+aclass +acquire-release +aes +armv8.2-a +cortex-x1 +crc +crypto +d32 +db +dotprod +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +hwdiv-arm +mp +neon +perfmon +ras +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8.1a +v8.2a +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization
arm-cortex-x1c	+aclass +acquire-release +aes +armv8.2-a +cortex-x1c +crc +crypto +d32 +db +dotprod +dsp +fp-armv8 +fp-armv8d16 +fp-armv8d16sp +fp-armv8sp +fp16 +fp64 +fpregs +fpregs16 +fpregs64 +fullfp16 +hwdiv +hwdiv-arm +mp +neon +perfmon +ras +sha2 +thumb-mode +thumb2 +trustzone +v4t +v5t +v5te +v6 +v6k +v6m +v6t2 +v7 +v7clrex +v8 +v8.1a +v8.2a +v8m +vfp2 +vfp2sp +vfp3 +vfp3d16 +vfp3d16sp +vfp3sp +vfp4 +vfp4d16 +vfp4d16sp +vfp4sp +virtualization

 © 2019-2024 SEGGER Microcontroller GmbH