# Introduction to the Armv8-M Architecture and its Programmers Model

Version 1.1

# User Guide

**Non-Confidential**

**Issue 01**

107656_0101_01_en

# Introduction to the Armv8-M Architecture and its Programmers Model
**User Guide**

## Release information

### Document history

| Issue | Date | Confidentiality | Change |
|---|---|---|---|
| 0100-01 | 4 November 2022 | Non-Confidential | First release |
| 0101-01 | 19 July 2023 | Non-Confidential | Second release |

## Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at https://www.arm.com/company/policies/trademarks.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

## 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

# 1. Introduction to Armv8 architecture and architecture profiles

The Armv8 architecture has several different profiles. These profiles are variants of the architecture that target different markets and use cases. The Armv8-M architecture is one of these architecture profiles.

Arm defines three architecture profiles: Application (A), Real-time (R), and Microcontroller (M).

The A profile:

- Supports the AArch64 or AArch32 Execution states
- Supports the A64, A32, and T32 instruction sets
- Supports a Virtual Memory System Architecture (VMSA) based on a Memory Management Unit (MMU)

The R profile:

- Supports the AArch64 or AArch32 Execution states
- Supports the A64, or A32 and T32 instruction sets
- Supports a Protected Memory System Architecture (PMSA) with optional support for VMSA at stage 1

The M profile:

- Implements a programmers' model designed for low-latency interrupt processing, with hardware stacking of registers and support for writing interrupt handlers in high-level languages
- Supports the T32 instruction set
- Supports a Protected Memory System Architecture (PMSA)

# 2. Introduction to Armv8-M architecture

The Armv8-M architecture defines many aspects of a Cortex-M processor's behavior, including the following:

- Programmers' model
- Instruction set
- Exception model
- Memory model
- Debug components

Armv8-M is a 32-bit architecture, which evolved from the Armv7-M and Armv6-M architectures. Armv8-M supports a subset of the T32 (Thumb) instruction set architecture. The T32 instruction set contains 16-bit and 32-bit instructions. All T32 instructions from the Armv7-M and Armv6-M architectures are supported in Armv8-M. This means that Armv8-M is backward compatible with both Armv7-M and Armv6-M.

The Armv8-M architecture registers, data operations, and addresses are all 32-bit. Although the Armv8-M architecture is 32-bit, it also supports data types of various sizes such as 8-bit, 16-bit, and a limited set of 64-bit operations. The Armv8.1-M optionally supports 128-bit operations as well.

The 32-bit physical address provides 4GB of address space, which is architecturally pre-defined into several regions with different memory attributes. Some portions of the memory space are used by the internal components of the processor core, such as programmable registers for the following:

- Nested Vectored Interrupt Controller (NVIC)
- SysTick timer
- Memory Protection Unit (MPU)
- System Control Block (SCB)
- Debug components

The rest of the memory space is utilized by chip designers. Architecturally, there is no restriction on the memory technology that can be connected to the systems, so products from different chip vendors can all have different types of memories and peripherals.

The following table shows the different generations of Cortex-M processors and the architectures they are built on:

**Table 2-1: Cortex-M processor generations**

| Generation | Processors |
|---|---|
| Armv7-M generation of Cortex-M processors | Cortex-M3, Cortex-M4, Cortex-M7 |
| Armv6-M generation of Cortex-M processors | Cortex-M0, Cortex-M0+, Cortex-M1 (FPGA) |

| Generation | Processors |
|---|---|
| Armv8-M generation of Cortex-M processors | Cortex-M23, Cortex-M33, Cortex-M35P, Cortex-M55, Cortex-M85 |

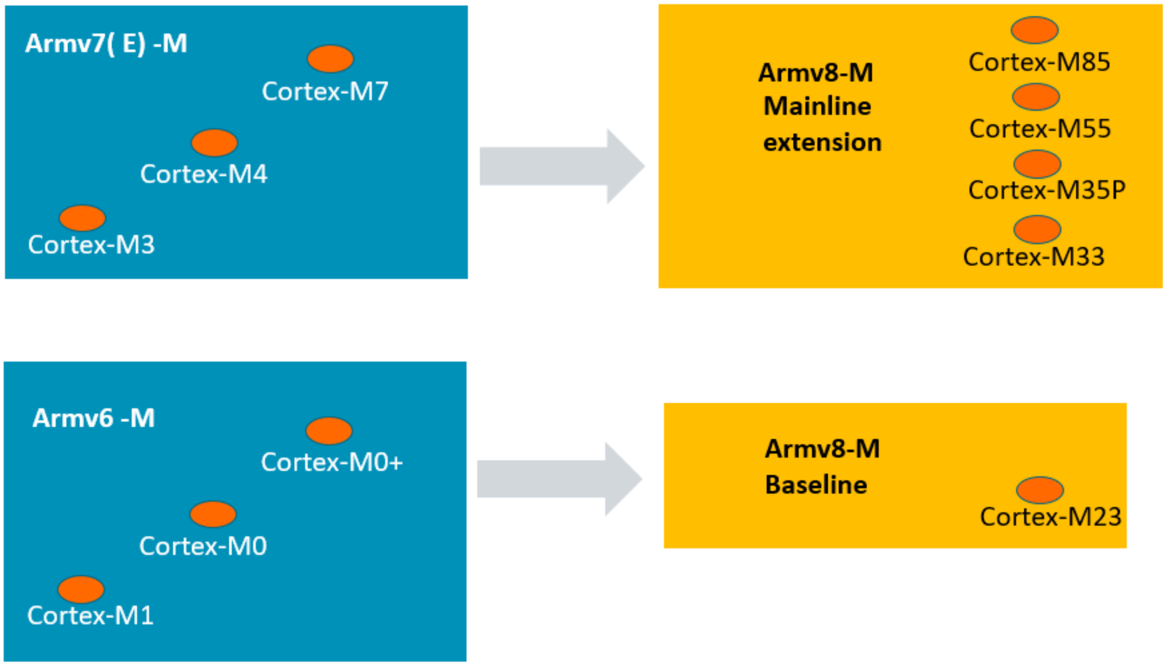
## 2.1 Baseline and Mainline

The Armv8-M architecture contains a baseline architecture feature set and supports several optional extensions to Armv8-M.

- Armv8-M Baseline: The simplest Armv8-M implementation, without any of the optional extensions, is a Baseline implementation.
- Armv8-M Main Extension: An Armv8-M implementation with the Main Extension is also referred to as a Mainline implementation. The Armv8-M Mainline implementation is the Armv8-M Baseline plus the Main Extension.

All Armv8-M implementations, both Baseline and Mainline, are compatible with Armv6-M. However, only Armv8-M Mainline implementations are compatible with Armv7-M. This is shown by the following diagram:

**Figure 2-1: Evolution of architectures for Cortex-M processors**



The Armv8-M architecture supports various optional extensions. The key optional extensions and their abbreviations include the following:

**M**

Main Extension

**FP**

Floating-point Extension. Enables support for the floating-point unit in an implementation.

**MVE**

M-Profile Vector Extension. Enables support for the features that are provided by the Armv8.1 M-Profile Vector Extension (MVE). Armv8-M MVE is also referred to as Arm Helium Technology for Armv8-M.

**MPU**

Memory Protection Unit. Enables support for the Memory Protection Unit in an implementation.

**DSP**

Digital Signal Processing Extension. Enables a range of instructions for digital signal processing in an implementation.

**DB**

Debug Extension. Enables additional debug features in an implementation.

**S**

Security Extension. The Armv8-M Security Extension is also referred as TrustZone Technology for Arm Cortex-M processors.

**RAS**

Reliability, Availability, and Serviceability Extension. Enables RAS support in an implementation.

## 2.2 Architecture and micro-architecture

The difference between architecture and micro-architecture is as follows:

### 2.2.1 Architecture

The details of the architecture used by Cortex-M processors are defined in the Armv8-M Architecture Reference Manual. For Armv8-M processors, the Armv8-M Architecture Reference Manual provides the specification of the programmer's model, instruction set, exception model, security architecture and debug architectures. The set of rules outlined in the Armv8-M Architecture Reference Manual outlines the behaviors of each instruction and the support available for debug tools, but not the details of how the processors are implemented.

### 2.2.2 Micro-architecture

Each Cortex-M processor has its own implementation level details. For example, the number of pipeline stage, the architectural features supported, the bus protocol used on the processor. High level specifications of the processors are detailed in Technical Reference Manuals. For example, the

Cortex-M33 Technical Reference Manual provides the details of the Cortex-M33 processor. Refer to the Cortex-M33 Technical Reference Manual for more details.

## 2.3 Compatibility with Armv6-M and Armv7-M

While the Armv8-M architecture is similar to Armv6-M and Armv7-M architecture, enabling most applications developed on previous architecture to run on the Armv8-M architecture, it also offers improvements over previous M-Profile architectures in the following areas:

- The optional Security Extension.
- An improved, optional, Memory Protection Unit (MPU) programmers' model.
- Alignment with Armv8-A and Armv8-R memory types.
- Stack pointer limit checking.
- Improved support for multi-processing (Exclusive access support).
- Better alignment with C11 and C11++.

Some of these Armv8-M features might impact the RTOS design.

# 3. Getting started with Armv8-M-based systems

This chapter gives a brief introduction about available platform, compiler, tools, and software support available for Armv8-M-based systems.

## 3.1 Hardware platforms and simulation models

There are several platforms and simulation models available for Armv8-M-based systems, including the following:

**Fast Models and Fixed Virtual Platforms (FVPs)**

An FVP is a virtual development platform built with Arm Fast Models for software development without a physical board. An FVP can be used standalone from a command-line interface.

Some FVPs are packaged as part of software development tools like Arm Development Studio and Keil MDK. These toolkits provide connection dialogs to allow the user to connect to the FVP through an IDE.

The Fast Models tool provides an environment to design and create custom virtual platforms, like FVPs, for early software development. The Fast Models tool provides different types of ready-made M-profile Fast Models. Refer to Fast Models on Arm Developer for more details. Note that Fast models and FVPs are functional models only and therefore they are not cycle accurate.

**RTL simulators from Arm EDA tool vendors**

See the Release Note and Integration and Implementation information for your Armv8-M implementation for further information on RTL simulator support. For example, if you are using a Cortex-M55 processor, then refer to the Cortex-M55 Release Note and Arm Cortex-M55 Processor Integration and Implementation Manual. Note that these documents are confidential and are only available to licensees.

**Arm MPS3 prototyping board**

The Arm MPS3 platform provides a way to load pre-built Arm subsystem images into its FPGA. For more information about the Arm MPS3 platform, see MPS3 FPGA Prototyping Board on Arm Developer.

**Arm Virtual Hardware (AVH)**

The AVH provides multiple Arm model platforms for developers to verify and validate embedded and IoT applications during software design cycle, see arm Virtual Hardware for more details.

**Arm IP Explorer**

The Arm IP Explorer is a cloud-based platform that can used by hardware engineers designing Arm-based systems. Helps create high level SoC design and evaluate IP compatibity. Ideal for SoC architects and system integrators, see Arm IP Explorer for more details.

**Third-party platforms**

Arm works closely with its partners who license Arm technology. Arm partners who have licensed Cortex-M processor(s) Intellectual Property (IP) for design typically develop their own platforms. Such platforms may be publicly available.

## 3.2 Arm Compiler for Embedded

Arm Compiler for Embedded is a component of Arm Development Studio and Arm Keil MDK. Alternatively, you can use Arm Compiler for Embedded as a standalone product.

Arm Compiler for Embedded combines the optimized tools and libraries from Arm with a modern LLVM-based compiler framework. The components in Arm Compiler for Embedded are:

**armclang**

The compiler and integrated assembler that compiles C, C++, and GNU assembly language sources.

**armasm**

The legacy assembler. Use armasm only for Arm-syntax assembly code.

**armlink**

The linker combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program.

**fromelf**

The image conversion utility can convert Arm ELF images to binary formats. It can also generate textual information about the input image, such as its disassembly, code size, and data size.

**Arm C libraries**

Arm C libraries provide an implementation of library features as defined in C standards.

For detailed information on Arm Compiler for Embedded please read the following:

- Arm Compiler for Embedded Reference Guide
- Arm Compiler for Embedded User Guide

### 3.2.1 Application development

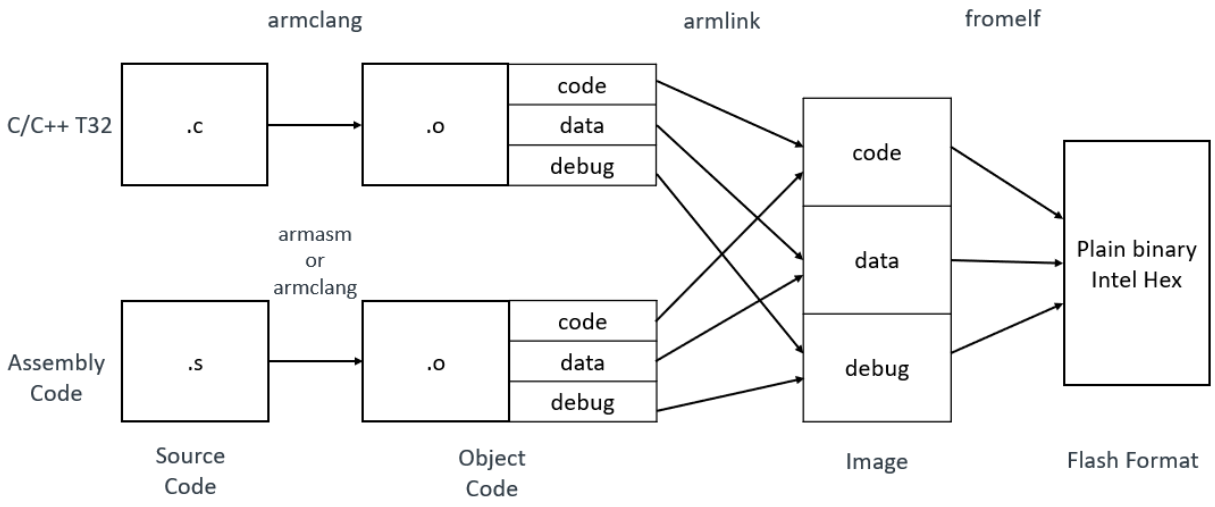
A typical application development flow might involve the following:

- Developing C/C++ source code for the main application (`armclang`).
- Developing assembly source code for near-hardware components, such as interrupt service routines (`armclang`, or `armasm` for legacy assembly code).
- Linking all objects together to generate an image (`armlink`).

- Converting an image to flash format in plain binary, Intel Hex, and Motorola-S formats (`fromelf`).

The following figure shows how development of a typical application uses the compilation tools:

**Figure 3-1: Compilation Tools Flow**

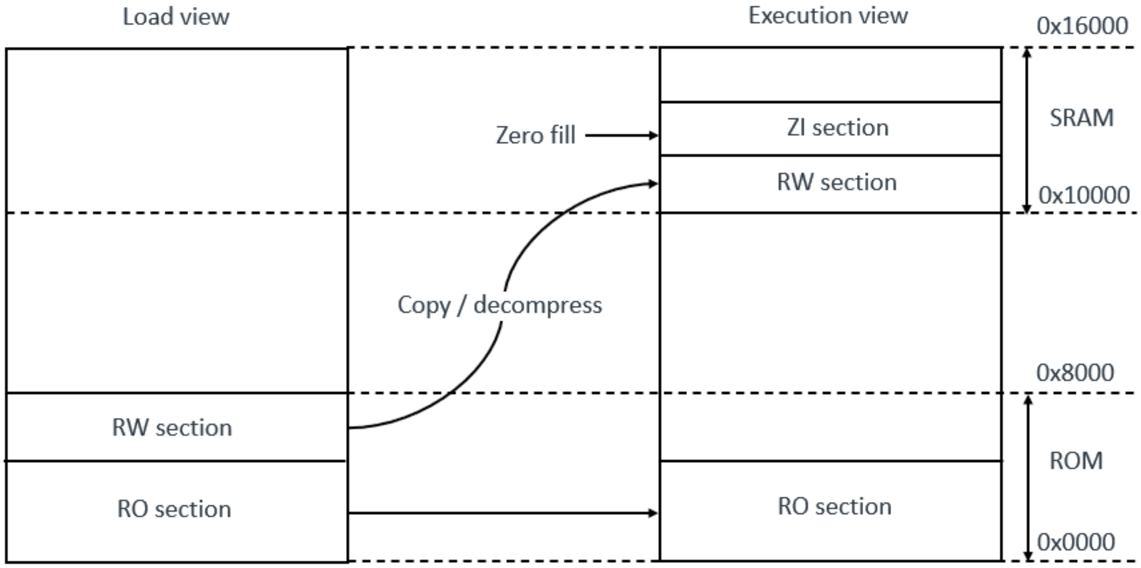


## 3.2.2 Scatter-loading images with simple memory map

For images with a simple memory map, you can specify the memory map using only linker command-line options, or with a scatter file.

The following figure shows a simple memory map:

**Figure 3-2: Scatter File Format**



The following example shows the corresponding scatter-loading description that loads the segments from the object file into memory:

```
LOAD_ROM 0x0000 0x8000       ; Name of load region (LOAD_ROM),
                            ; Start address for load region (0x0000),
                            ; Maximum size of load region (0x8000)
{
    EXEC_ROM 0x0000 0x8000   ; Name of first exec region (EXEC_ROM),
                            ; Start address for exec region (0x0000),
                            ; Maximum size of first exec region (0x8000)
    {
        * (+RO)             ; Place all code and RO data into
                            ; this exec region
    }
    SRAM 0x10000 0x6000      ; Name of second exec region (SRAM),
                            ; Start address of second exec region (0x10000),
                            ; Maximum size of second exec region (0x6000)
    {
        * (+RW, +ZI)        ; Place all RW and ZI data into
                            ; this exec region
    }
}
```

The maximum size specifications for the regions are optional. However, if you include them, they enable the linker to check that a region does not overflow its boundary.

Apart from the limit checking, you can achieve the same result with the following linker command-line:

```
armlink --ro_base 0x0 --rw_base 0x10000
```

### 3.2.3 Tailoring the image memory map to your target hardware

You can use a scatter file to define a memory map, giving you control over the placement of data and code in memory. In your final embedded system, without semihosting functionality, you are unlikely to use the default memory map. Your target hardware usually has several memory devices located at different address ranges. To make the best use of these devices, you must have separate views of memory at load and runtime.

The following shows how to use a scatter file by running the `armlink` command with the "–scatter" option:

```
armlink --scatter scatter.scat file1.o file2.o
```

Scatter-loading defines two types of memory regions:

- Load regions containing application code and data at reset and load-time.
- Execution regions containing code and data when the application is executing. One or more execution regions are created from each load region during application startup.

A single code or data section can only be placed in a single execution region. It cannot be split.

For more details on tailoring your target memory map with stack, heap, and location of target peripherals, refer to the Embedded Software Development section in the Arm Compiler for Embedded User Guide.

## 3.3 Debug tools support

Designed for the Arm architecture, Arm Development Studio (Arm DS) and Keil MDK is the most comprehensive embedded C/C++ dedicated software development solution which supports debug for Cortex–M CPUs. Its components include the following:

- Arm Compiler for Embedded 6 for compiling bare-metal embedded applications. Includes support for the latest Arm architectures.
- Arm Compiler for Embedded FuSa to accelerate the building of safety critical code and simplify TÜV SÜD certification process.
- Complete library of reference Fixed Virtual Platforms (FVPs) along with pre-built examples.
- Entitlement to Keil MDK Professional Edition is included in Silver, Gold, and Platinum editions.

## 3.4 Common Microcontroller Software Interface Standard (CMSIS)

As the complexity of embedded systems increases, the compatibility and portability of software code becomes even more important. Having a reusable software often helps in reducing the development time for subsequent projects. To allow a high level of compatibility between software products and to improve software portability and reusability, Arm has worked with several microcontroller tool vendors and software solution providers to develop the CMSIS - a common software framework for Cortex-M processors and Cortex-M microcontroller products.

CMSIS-Core is part of the Cortex Microcontroller Software Interface Standard (CMSIS) and provides a standardized API for different aspects of software development for the Cortex-M devices, including the following:

- Startup and initialization code templates.
- Processor core instruction intrinsics.
- Processor core peripheral functions and macros.
- Device-specific system clock and peripheral macros and functions.

More details on CMSIS-Core source code and documentation are available from the following CMSIS GitHub repository:

- github.com/ARM-software/CMSIS_5

Few of the popular compilers that CMSIS-Core supports are listed below:

- Arm Compiler (version 5 and later)
- GNU Arm Embedded Toolchain
- IAR C/C++ Compiler

Arm Compiler is available as part of the following products:

- Arm Development Studio (Arm DS)
- Keil Microcontroller Development Kit (Keil MDK)

The following table lists some of the CMSIS-CORE files in a typical Cortex-M33 software project:

| File | Description |
|---|---|
| `<device>.h` | Definition of constants, peripheral device register definitions required by CMSIS-CORE |
| `core_cm33.h` | Definition of registers for processor peripherals such as NVIC, SysTick Timer, System Control Block (SCB) |
| `cmsis_compiler.h` | Enables selection of compiler-specific header files |
| `cmsis_armclang.h` | Provides intrinsic functions and core register access functions |
| `cmsis_armcc.h` | Provides intrinsic functions and core register access functions |
| `cmsis_version.h` | CMSIS version information |
| `system_<device>.h` | Header file for functions implemented in `system_<device>.c` |
| `system_<device>.c` | 1. System initialization function called `void SystemInit(void)`<br>2. Definition of variable for clock `SystemCoreClock` |

## 3.5 Procedure Call Standard for Arm Architecture (AAPCS)

When a function is written using assembly language and needs to interact with other C codes, there is a range of requirements that need to be followed to allow the interface between software functions to work. These requirements are captured in the Procedure Call Standard for Arm Architecture (AAPCS). Some of the main areas covered by the AAPCS are as follows:

**Register usage in function calls**

Defines the registers that are termed as Caller-saved and Callee-saved. For example, a function call should retain the values in R4-R11. If these registers need to be changed during function call execution, then they should be saved and restored before ending the function call.

**Passing parameters to functions**

The AAPCS defines the registers to use to pass parameters to function calls. The exact registers that are used depends on the number and size of the parameters being passed. For example, in a simple case where there are two integer parameters being passed to a function, then the AAPCS defines that the R0 and R1 registers could be used to pass the two parameters.

**Stack alignment**

If an assembly function needs to call a C function, it should ensure that the current selected stack pointer points to a doubleword-aligned address location.

For more details, see Procedure Call Standard for Arm Architecture.

## 3.6 Arm C Language Extensions (ACLE)

The Arm architecture includes features that go beyond the set of operations available to C/C++ programmers. The intention of the Arm C Language Extensions (ACLE) is to allow the creation of applications and middleware code that is portable across compilers and across Arm architecture variants while fully utilizing the advanced features of the Arm architecture.

The ACLE standardizes the intrinsics to access Arm instructions which do not map directly to C operators generally either for optimal implementation of algorithms or for accessing special system-level features. The ACLE incorporates some language extensions introduced in the GCC C compiler. The ACLE extends some of the guarantees of C, allowing assumptions to be made in source code beyond those permitted by Standard C.

The ACLE provides details on supported intrinsics, predefined macros, and attributes. Refer to the Arm C Language Extensions for more detailed explanations and information about corresponding architecture features.

# 4. Operational modes and states

The Armv8-M architecture has two operating states and two operating modes. Along with these operating modes and states, the Armv8-M architecture supports privileged and unprivileged access levels. The privileged access level can access all resources in the processor, while the unprivileged access level means some memory regions are inaccessible and a few operations cannot be used.

## 4.1 Operating states

The two operating states provided by Armv8-M are as follows:

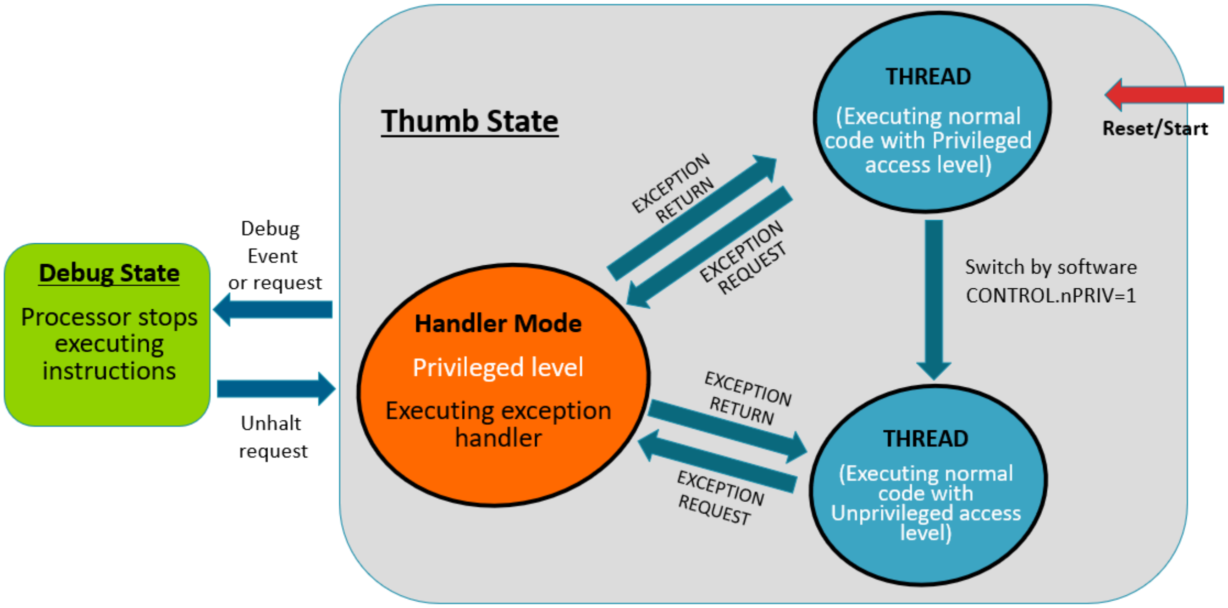
**Thumb state**

If the processor is running the software program code (Thumb instructions), it is in Thumb state. Armv8-M-based processors do not support A32 instruction set and there is no Arm state.

**Debug state**

When the processor is halted, for example by a debugger or by hitting a breakpoint, it enters Debug state and stops executing instructions.

**Figure 4-1: Operation states and modes**


Thumb State
THREAD
(Executing normal code with Privileged access level)
Reset/Start
EXCEPTION RETURN
EXCEPTION REQUEST
Switch by software CONTROL.nPRIV=1
Debug State
Processor stops executing instructions
Debug Event or request
Unhalt request
Handler Mode
Privileged level
Executing exception handler
EXCEPTION RETURN
EXCEPTION REQUEST
THREAD
(Executing normal code with Unprivileged access level)

Debug state is only used for debugging operations. In Debug state, processor execution is halted and no instructions can be executed by processor. This state is entered by a halt request from the debugger, or by debug events generated from debug components in the processor. This state allows the debugger to access or change the processor register values, peripheral registers, and system memory.

## 4.2 Operating modes

The two operating modes provided by Armv8-M are as follows:

**Thread mode**

When executing application code, the processor can be either in privileged access level or unprivileged access level. This is controlled by the special-purpose CONTROL register. Refer to Registers for more information about special-purpose registers. On reset, the processor enters privileged Thread mode in Thumb state. When the processor is executing in Thread mode, it has an option to switch to using a separate banked Stack Pointer (SP).

**Handler mode**

When executing an exception handler such as Interrupt Service Routine (ISR), the processor will be in handler mode. When in handler mode, the processor always has privileged access level.

Note

By default, the Cortex-M processors start in privileged Thread mode and in Thumb state. In many simple applications, there is no need to use the unprivileged Thread model and the banked SP at all.

## 4.3 Privileged and unprivileged execution

In privileged execution, software can use all instructions and has access to all resources. Privileged software can write to the CONTROL register to change from privileged to unprivileged for software execution in Thread mode. However, it cannot switch itself back from unprivileged to privileged.

Within a single security state, only a transition to handler mode can change from unprivileged to privileged execution. In unprivileged mode, software has limited access to instructions that change processor state. It cannot access the system timer, NVIC, or System Control Block, and has restricted access to memory or peripherals that are marked with privileged access only.

The separation of privileged and unprivileged execution allows system designers to give different permissions to different parts of the system. For example, critical sections like the OS kernel can be privileged, while user application tasks are unprivileged. Besides the differences in memory access permissions and access to several special instructions, the programmers' model of privileged access level and unprivileged access level are almost the same.

## 4.4 Secure and Non-secure states

When the optional Security Extension is implemented in a processor, there are two new states, the Secure state and the Non-Secure state. The existing thread and handler modes are duplicated between the two security states:

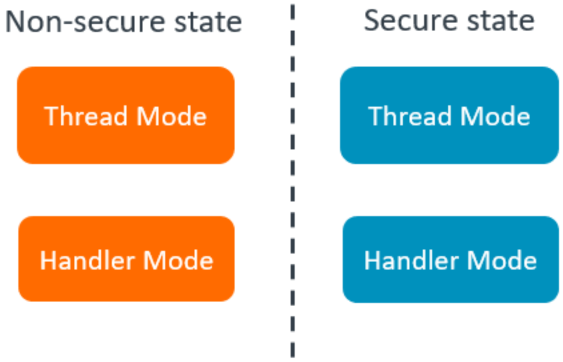
### Secure state

When the processor is in Secure privileged state, it can access all resources. When the core is in Secure state, it can access both Secure and Non-secure memory. If the Security Extension is implemented in an Armv8-M-based processor, then the processor starts up in Secure, privileged, thread mode.

### Non-secure state

When the processor is in Non-secure privileged state, it can gain access to resources subject to security access permissions defined by Secure software (SAU and IDAU). When the core is in Non-secure state, it can only access Non-secure memory. If the Security Extension is not implemented in an Armv8-M-based processor, then there is no Secure state and the processor starts up in Non-secure privileged thread mode.

**Figure 4-2: Security states and modes**



Note that Security states and privilege modes are orthogonal to each other. Both Secure and Non-Secure state supports privileged and unprivileged levels.

More details on Secure and Non-secure states along with privilege level and operating modes are available in the Armv8-M Security Extension User Guide.

# 5. Registers

Arm is a load/store architecture. When processing data, the processor loads the data into a register from memory, performs data processing operations, and then optionally stores the result back into memory. Armv8-M supports 32-bit registers which enables 32-bit data processing. There are various types of registers supported in the Armv8-M architecture to enable 32-bit data processing. If the Floating-point (FP) or Digital Signal Processing (DSP) extensions are included, then there are some operations that can perform 64-bit data processing.

The various types of registers within the Armv8-M architecture include the following:

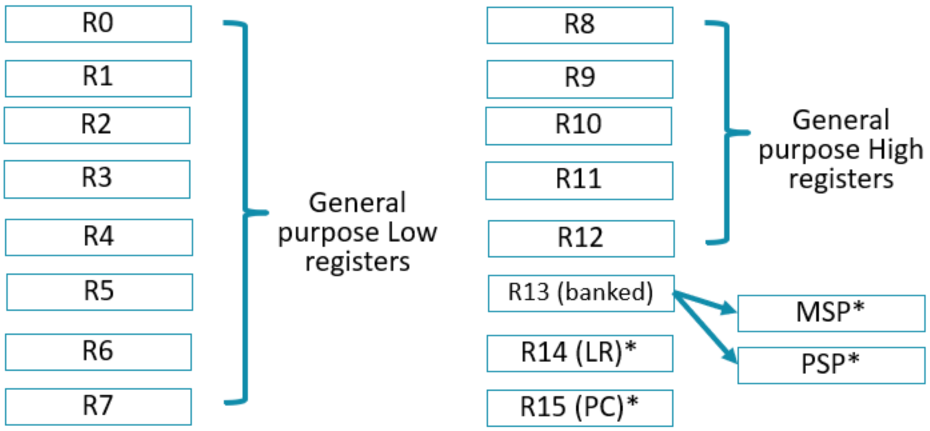
1. General-purpose registers
2. Special-purpose registers
3. Memory-mapped registers

The following sections describe each of these different types of registers available within the Armv8-M Architecture.

## 5.1 Registers in the register bank

There are sixteen 32-bit registers in the register bank, referred to R0-R15. R0-R12 are general-purpose registers used by most of the instructions. R13-R15 have designated usage.

**Figure 5-1: Registers**



*MSP – Main Stack Pointer
*PSP – Process Stack Pointer
*LR – Link Register
*PC – Program Counter

### 5.1.1 R0-R12

Registers R0 to R12 are general-purpose registers. The first eight (R0-R7) are also called low registers. Due to the limited number of bits in instruction opcodes, many 16-bit instructions can only access the low registers. The high registers (R8-R12) can be used with 32-bit instructions and with some 16-bit instructions like the `MOV` instruction. The initial values of R0 to R12 are **UNKNOWN** out of reset.

### 5.1.2 R13, Stack Pointer (SP)

R13 is the Stack Pointer (SP). It is used for accessing the stack memory, for example with `PUSH` and `POP` instructions. When the processor pushes new data onto the stack, it decrements the stack pointer and then writes the data to the memory location. Physically, there can be either two or four different stack pointers.

If the Security Extension is not implemented, then there are two stack pointers:

**Main stack pointer, commonly referred to as MSP or SP_Main**

The MSP is the default stack pointer used at reset, and is used for all exception handling.

**Process stack pointer, commonly referred as PSP or SP_Process**

The PSP is the alternative stack pointer that can only be used in thread mode, and is usually used for application tasks of the operating system (OS).

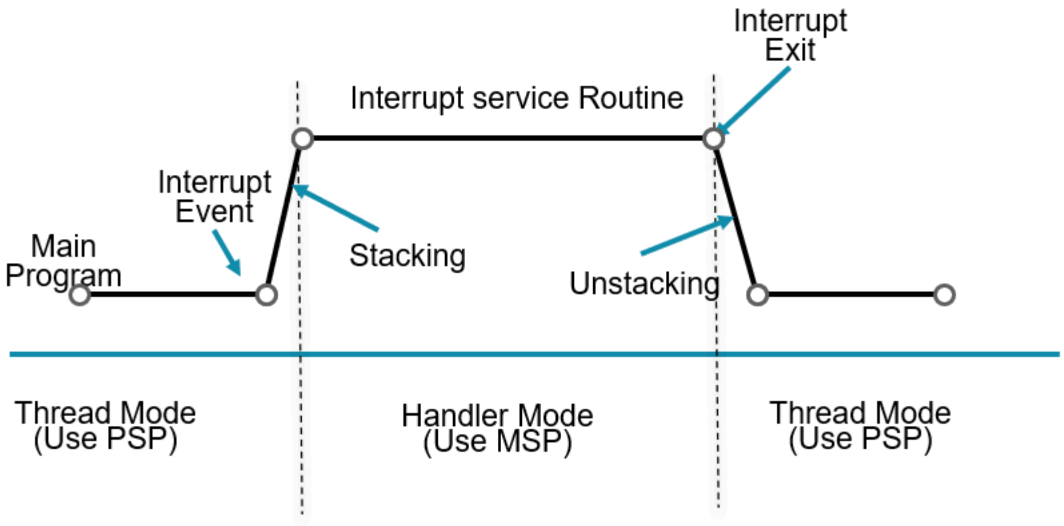
Stack pointer selection is determined by the special register called CONTROL. The SPSEL field in the CONTROL register can be programmed to select between stack pointers for thread mode stack operations as follows:

- Set SPSEL to 0 to select MSP.
- Set SPSEL to 1 to select PSP.

In normal program execution, only one of these stack pointers will be active. Though both MSP and PSP are 32-bit registers, the lowest two bits of the stack pointers are always zero, and writes to these two bits are ignored. The stack memory operations such as `PUSH` and `POP` are always 32-bit. Refer to Stack memory for more information about `PUSH` and `POP` operations with stack memory.

If an operating system is used, the stack for each of the application threads will be separated from each other. The PSP enables application threads to switch contexts without affecting the stack used by privileged code. The stack selected on an exception return is based on the SPSEL bit in the EXC_RETURN payload value which is automatically stacked on exception entry. This is shown in the following diagram:

**Figure 5-2: PSP stacking and unstacking**



Warning

Software must never place any data below the current stack pointer position. Stacking on exception entry can occur at any time (for example, in response to an interrupt) and can overwrite any data below the stack pointer.

If the Security Extension is implemented in a system, then there are four stack pointers:

- Main Stack Pointer for Non-secure state (MSP_NS)
- Process Stack Pointer for Non-secure state (PSP_NS)
- Main Stack Pointer for Secure state (MSP_S)
- Process Stack Pointer for Secure state (PSP_S)

The stack pointers are banked when the Security Extension is implemented. In both Secure and Non-secure states, the processor implements the main stack and the process stack, with a pointer for each held in independent registers. The _S and _NS suffixes identify whether the stack pointer is for the Secure state or Non-secure state.

The following table shows which stack pointer is banked between Secure and Non-secure states:

| Stack Pointer | Secure State | Non-secure State |
|---|---|---|
| Main Stack Pointer | MSP_S | MSP_NS |
| Process Stack Pointer | PSP_S | PSP_NS |

#### 5.1.2.1 Accessing stack pointers

The current stack pointer selected by CONTROL.SPSEL can be accessed as SP/R13 with many instructions. Here is an example:

```
// When CONTROL.SPSEL = 0;
ADDW R0,SP,#1 ; R0 = MSP+1

// When CONTROL.SPSEL = 1;
SUBW R2,SP,#1 ; R2 = PSP-1;
```

The MSP and PSP registers are accessed using the special register access instructions `MRS` and `MSR`:

```
MRS <register>, <special_reg>: Read special register into general-purpose register
MSR <special_reg>, <register>: Write to special register from general-purpose
 register
```

The following examples show how to access the MSP and PSP registers:

```
MRS R5, MSP ; R5 = MSP, read MSP into R5 register.
MSR PSP, R7 ; PSP = R7, write R7 value into PSP.
```

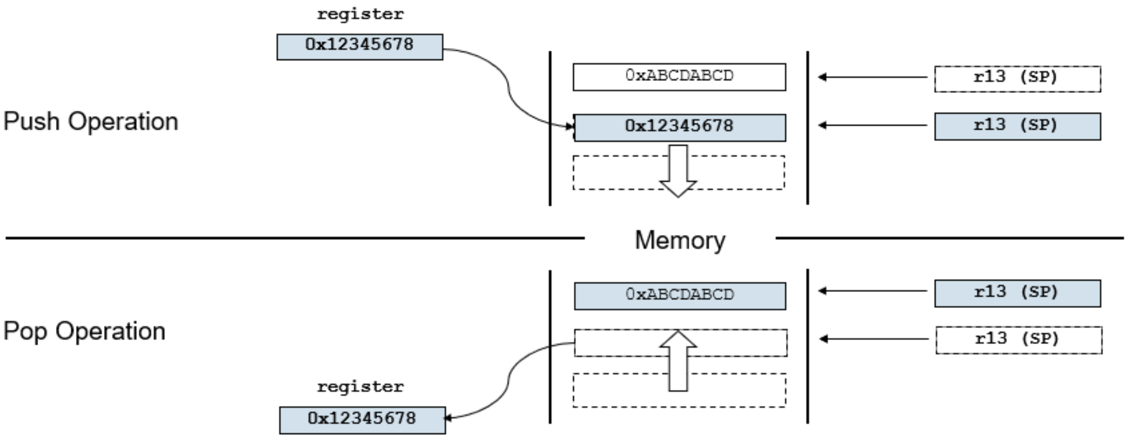
Out of reset, the processor automatically initializes the MSP for the security state being entered by reading the vector table offset `0x0`. The PSP stack pointer is not initialized immediately out of reset. Privileged software should initialize PSP if needed. The CMSIS-CORE software framework provides functions for stack pointer access. The following table summarizes these stack pointer access functions:

| CMIS-CORE function name | Description |
|---|---|
| `__get_MSP(void)` | Gets the value of MSP. |
| `__get_PSP(void)` | Gets the value of PSP. |
| `__set_MSP(uint32_t topofstack)` | Sets the value of MSP. |
| `__set_PSP(unit32_t topofstack)` | Sets the value of PSP. |

#### 5.1.2.2 Stack memory

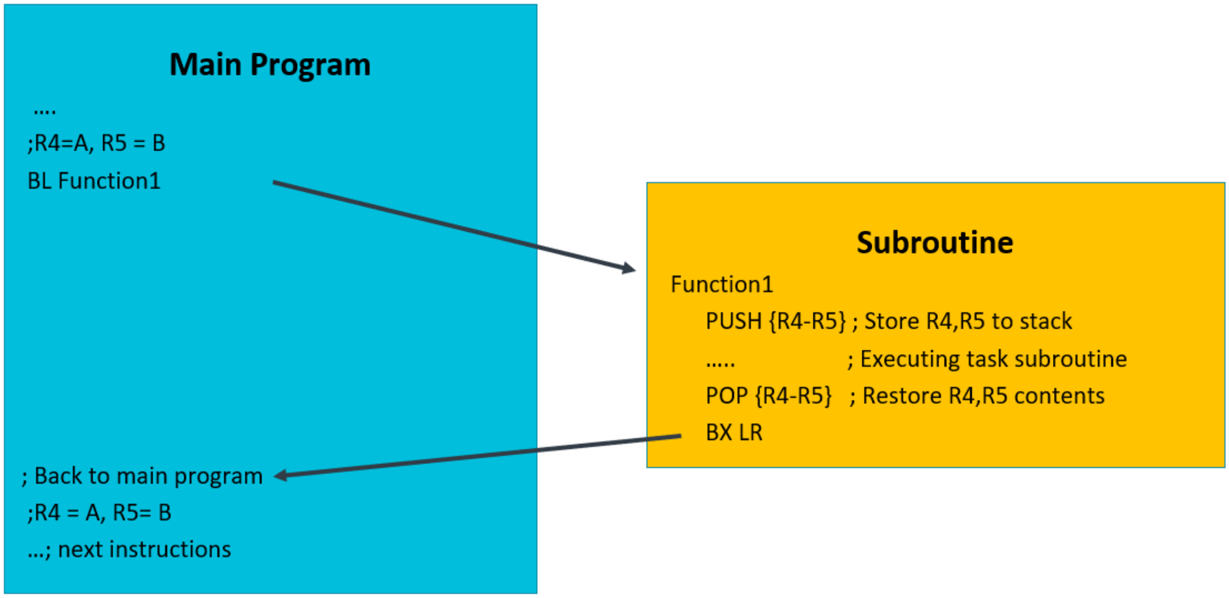
Arm Cortex-M processors have dedicated stack pointer (R13) hardware for stack operations. Stack is a memory usage mechanism that allows a portion of memory to be used as a Last-In-First-Out (LIFO) data storage buffer. Arm Cortex-M processors use the main memory address space (such as RAM) for stack memory operations. They have a `PUSH` instruction to store data in the stack and a `POP` instruction to retrieve data. The currently selected stack pointer is automatically adjusted for each `PUSH` and `POP` operation. Cortex-M processors use a full descending stack. This means that the stack pointer always points to the last stored data in the stack memory, and the stack pointer pre-decrements for each new PUSH operation.

**Figure 5-3: Push and pop operations**



The most common use for PUSH and POP instructions is to save the contents of register banks when a function or subroutine call is made. At the beginning of the function call, the contents of some of the registers can be saved to the stack using a PUSH instruction and then restored to their original values at the end of the function using a POP instruction. For example, in the figure below, a simple function named Function1 is called from the main program. Since Function1 needs to use and modify R4 and R5 for data processing, and these registers hold values that the main program needs later, they are saved to the stack using a PUSH instruction and then restored using a POP at the end of Function1. This way, the program code that called the function will not lose any data from R4 and R5 and can continue to execute. Since the registers are 32 bits, each stack PUSH and stack POP operation transfers at least 1 word (4 bytes) of data.

**Figure 5-4: Banking registers on a function call**



```
....
;R4=A, R5 = B
BL Function1

; Back to main program
;R4 = A, R5= B
...; next instructions
```

```
Function1
    PUSH {R4-R5} ; Store R4,R5 to stack
    .....               ; Executing task subroutine
    POP {R4-R5}   ; Restore R4,R5 contents
    BX LR
```

Note

Software Best Practices:

When saving and restoring registers, for each `PUSH` (store to memory) operation, there must be a corresponding `POP` (read from memory) and the address of the `POP` should match that of the `PUSH` operation.

## 5.1.3 R14, Link Register (LR)

R14 is also called the Link Register (LR). This holds the return address when calling a function or subroutine.

At the end of a function or subroutine, the callee function can return to the calling function and resume by loading the value of LR into the Program Counter (PC). When a function or subroutine call is made, the value of LR is updated automatically.

If a callee function needs to call another function, it needs to save the value of LR in the stack, otherwise the current value in LR will be lost when the function call is made. During exception entry, the LR is updated automatically to a special EXC_RETURN (exception return) payload value. At the end of the exception handler the software branches to the EXC_RETURN value in LR. The processor detects a branch to this special address and triggers the exception return process. This enables exception handlers to be written directly in C, without the need for an assembly wrapper around the body of the exception handler. Further details on the exception entry and exit process will be covered in the Armv8-M Exception Model User Guide.

### 5.1.4 R15, Program Counter (PC)

R15 is the Program Counter (PC). It is readable and writable. A read returns the current instruction address + 4 while writing to a PC (for example using data processing instructions) causes a branch operation.

Since all instructions must be aligned to halfword or word addresses, the Least Significant Bit (LSB) of the PC is always zero. However, when using branch and memory read instructions to update the PC, you will need to set the LSB of new PC values to 1 to indicate the Thumb state. If the LSB of PC is not set to 1, then it results in a UsageFault or HardFault fault exception.

In high-level programming languages, including C and C++, the compiler automatically sets the LSB in branch targets. Out of reset, the PC is initialized by the value from reset vector address location as a part of the hardware startup sequence.

All the registers from the register bank can be accessed (read or write) using debug software when the processor is in halted Debug state.

## 5.2 Special-purpose registers

In addition to the general-purpose registers, the Armv8-M architecture specifies a set of special-purpose registers.

For example, these special-purpose registers include control registers for interrupt handling, conditional flags for reflecting data processing results for arithmetical operations, and logical operations. The special-purpose registers are accessed using the special register access instructions `MRS` and `MSR`:

- `MRS <register>, <special_reg>`: Read special register into general-purpose register
- `MSR <special_reg>, <register>`: Write to special register from general-purpose register

In C programming, the CMSIS-CORE defines C functions for accessing special-purpose registers.

Note

In an Armv8-M-based system, special-purpose registers are physical registers available within the processor.

### 5.2.1 Program Status Registers

The Program Status Register is a 32-bit register and is subdivided into the following:

**Application Program Status Register (APSR)**

Contains various ALU flags which are required for conditional branches and instruction operations that need special flags, for example subtract with carry.

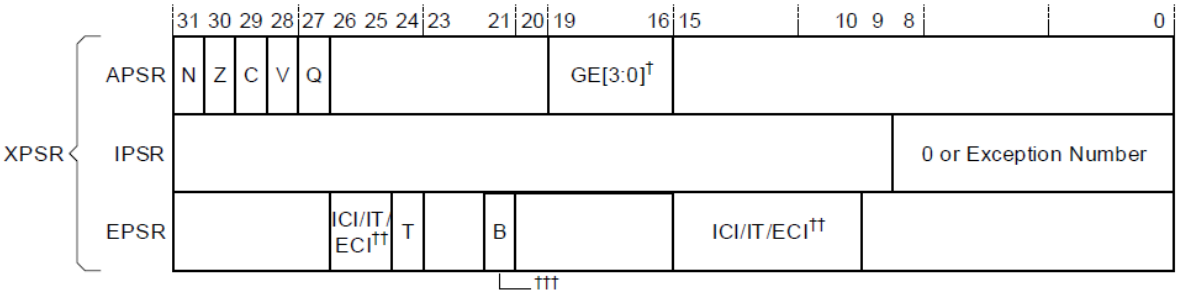
### Interrupt Program Status Register (IPSR)

Contains current interrupt or exception state information.

### Execution Program Status Register (EPSR)

Contains execution state information. The EPSR contains the state bit (T) and the execution state bits for either the If-Then (IT) instruction, or the Interruptible-Continuable Instruction (ICI) field for an interrupted load multiple or store multiple instruction.

### Figure 5-5: PSR register fields



† Reserved if the DSP Extension is not implemented
†† Reserved if the Main Extension is not implemented. ECI requires implementing the MVE Extension.
††† B requires implementing the PACBTI Extension.

These three registers can be accessed as one combined register, often referred as XPSR.

You can access the program status registers individually. For example,

```
MRS R0, APSR          : Read flag states into register R0
MRS R0, IPSR          : Read exception and interrupt states into register R0
MSR APSR, R0          : Write flag states
```

The following table shows the possible combinations of accessing XPSR.

| Symbol | Description |
|---|---|
| APSR | Application PSR only |
| EPSR | Execution PSR only |
| IPSR | Interrupt PSP only |
| IAPSR | Combination of APSR and IPSR |
| EAPSR | Combination of APSR and EPSR |
| IEPSR | Combination of IPSR and EPSR |
| PSR | Combination of APSR, IPSR and EPSR |

Note

Access restrictions:

The EPSR cannot be accessed by the software code using the `MRS` or `MSR` instructions. The contents can be viewed when the xPSR is saved and restored as a part of exception entry and exit process.

The IPSR is read-only and cannot be changed by the `MSR` instruction.

For more details on individual bit field descriptions, refer to the Armv8-M Architecture Reference Manual.

The CMSIS-CORE software framework provides functions for accessing Program Status Registers. The following table summarizes these functions for accessing PSR:

| CMIS-CORE function name | Usage details |
|---|---|
| `__get_APSR(void)` | Read APSR register |
| `__get_IPSR(void)` | Read IPSR register |
| `__get_xPSR(void)` | Read xPSR register |

## 5.2.2 Exception mask registers

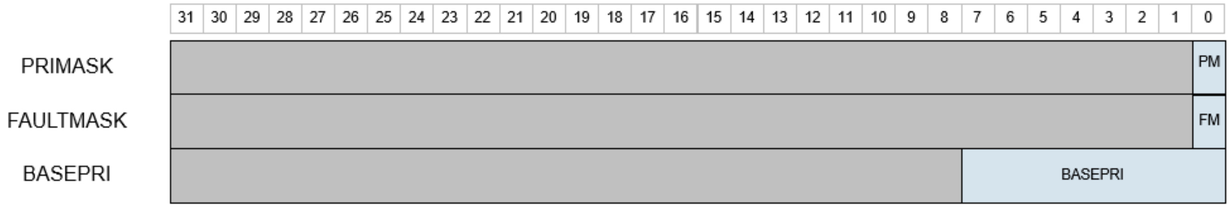
There are three types of exception and interrupt mask registers available in the Armv8-M architecture:

- 1-bit exception mask register, PRIMASK
- 1-bit fault mask register, FAULTMASK
- 8-bit base priority mask register, BASEPRI

Each exception (including interrupts) has a priority level. A smaller number is a higher priority and a larger number is a lower priority. These special-purpose mask registers are used to mask exceptions based on priority levels. These mask registers play a vital role in priority boosting methods for exception handling within the processor. An example of the use of these registers would be to disable exceptions when they might affect timing-critical tasks.

Out of reset, bit fields in these registers are set to zero. This means that masking or disabling of exceptions and interrupts is not active out of reset.

**Figure 5-6: Mask registers**



1. PRIMASK: The PRIMASK register is a 1-bit interrupt mask register. When set, it blocks all exceptions (including interrupts) apart from Non-Maskable Interrupts (NMI) and HardFault exceptions. This effectively means that the current exception priority level is 0. Note that 0 is the highest level for a programmable exception or interrupt.

2. FAULTMASK: Similar to the PRIMASK register, the FAULTMASK register also blocks all exceptions including HardFault exception. This effectively means that the current exception priority level is -1. Typically, NMI can preempt the program when the exception priority level is -1.
3. BASEPRI: To allow flexible interrupt masking, the Armv8-M architecture provides the BASEPRI register which masks exceptions and interrupts based on priority level. The width of the BASEPRI register depends on how many priority levels are implemented. When BASEPRI is set to 0, it means that it is disabled. When BASEPRI is set to a nonzero value, it blocks exceptions that have the same or lower priority level while allowing exceptions with a higher priority level.

In Armv8-M Baseline processors such as Cortex-M23, only PRIMASK is available. FAULTMASK and BASEPRI registers are not included in the Armv8-M Baseline architecture. More details on PRIMASK, FAULTMASK and BASEPRI registers are provided in the Armv8-M Exception Model User Guide.

`CPS` (Change Processor State) instructions can modify PRIMASK and FAULTMASK registers. If you want to access BASEPRI you must use `MRS` and `MSR` instructions. Note that these special purpose registers can be modified only in a privileged mode.

```
CPSIE i                ; Enable exceptions by clearing PRIMASK
CPSID i                ; Disable exceptions by setting PRIMASK
CPSIE f                ; Enable exceptions by clearing FAULTMASK
CPSIE f                ; Disable exceptions by setting FAULTMASK

MRS R0,BASEPRI         ; Read the BASEPRI register value
MSR BASEPRI,R2         ; Write R2 value into BASEPRI register
MSR BASEPRI_MAX,R3     ; Write to BASEPRI only when R3 value
                       ; is higher priority than current BASEPRI
```

Note

The MSR instruction can also be used for a qualified version of BASEPRI called BASEPRI_MAX. If an MSR instruction is used to write to the BASEPRI_MAX register, then it will update the BASEPRI register only if the new value is higher priority than the current BASEPRI setting. If the BASEPRI value is already at a higher priority level than the new value, then BASEPRI register will remain unchanged.

For example, consider a situation where the BASEPRI register is configured with a priority value of 16. If the BASEPRI_MAX register is written with a priority value of 12 then the execution priority would increase to this new priority value 12, disabling additional exception priorities from 15 through 12. However, if the BASEPRI_MAX register is written with a priority value of 20, then the BASEPRI register would remain unchanged with a priority value of 16.

Writing to BASEPRI_MAX is useful when you want to ensure that the priority level is changed to a higher priority only when needed. This avoids the sequence of reading the current value of BASEPRI, comparing it with a priority level, and then writing a new value into BASEPRI register.

In C programming, CMSIS-CORE provides functions to access these mask registers, as follows:

| CMSIS-CORE function name | Usage |
|---|---|
| `__get_BASEPRI(void)` | Read BASEPRI register |
| `__get_PRIMASK (void)` | Read PRIMASK register |
| `__get_FAULTMASK (void)` | Read FAULTMASK register |
| `__set_BASEPRI(uint32_t basePri)` | Set a new value for BASEPRI |
| `__set_BASEPRI_MAX(uint32_t basePri)` | Assigns the given value to the Base Priority register only if the new value increases the BASEPRI priority level |
| `__set_PRIMASK(uint32_t priMask)` | Set a new value for PRIMASK |
| `__set_FAULTMASK(uint32_t faultMask)` | Set a new value for FAULTMASK |
| `__disable_irq (void)` | Disables interrupts by setting PRIMASK |
| `__enable_irq (void)` | Enables interrupts by clearing PRIMASK |

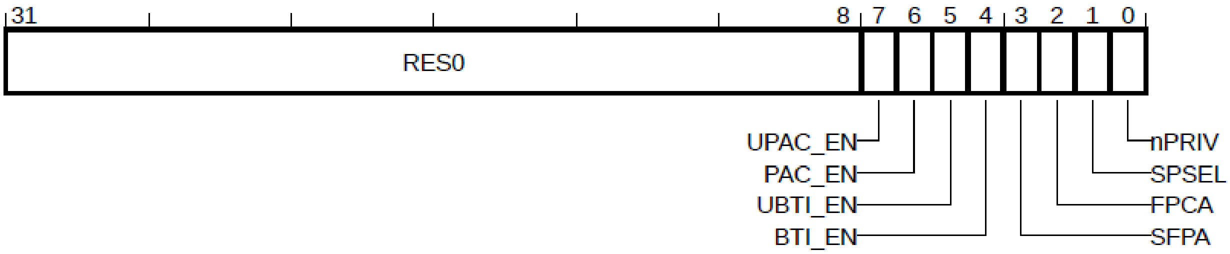
### 5.2.3 CONTROL register

The CONTROL register contains multiple bit fields depending on the features supported in an implementation. A CONTROL register can be any one of the following:

- 2-bit when none of the optional features such as the Floating-point Extension, MVE, and the Security Extension are implemented.
- 3-bit when either the Floating-point Extension or MVE are implemented, but other optional features are not implemented.
- 4-bit or 8-bit depending on the support of certain optional features.

The following diagram shows the CONTROL register bit assignments:

**Figure 5-7: The CONTROL register**



The CONTROL register can only be written in privileged state, but can be read by both privileged and unprivileged program software. When the Security Extension is implemented, some of the bit fields are banked between security states.

Out of reset, the value of the CONTROL register is 0, which means the following:

1. CONTROL[0] -> Program execution starts from Privileged thread mode (indicated by CONTROL.nPRIV = 0)

2. CONTROL[1] -> MSP is the current selected stack pointer (indicated by CONTROL.SPSEL = 0)
3. CONTROL[3:2] -> If the Floating-point Extension is implemented in a system, then the FPU (Floating-Point Unit) does not contain any active context data (indicated by CONTROL.FPCA = 0) and does not hold any secure data (indicated by CONTROL.SFPA = 0).
4. CONTROL[7:4] -> If the PACBTI Extension is implemented in a system, then the Pointer Authentication and Branch Target Identification Enable bits are set to 0 (CONTROL[7:4] = 4'b0000)

Privileged software can optionally write to the CONTROL register to do the following:

- Switch the stack pointer selection by CONTROL.SPSEL:
    - If this bit is 0, MSP is selected
    - If this bit is 1, PSP is selected
- Switch the privileged level by CONTROL.nPRIV:
    - If this bit is 0, switch to privileged thread mode
    - If this bit is 1, switch to unprivileged thread mode
- Enable Pointer Authentication Code (PAC):
    - If CONTROL.PAC_EN is set to 1, then Pointer authentication is enabled for privileged accesses.
    - If CONTROL.UPAC_EN is set to 1, then Pointer authentication is enabled for unprivileged accesses.
- Enable Branch Target Identification (BTI):
    - If CONTROL.BTI_EN is set to 1, then Branch Target Identification enabled for privileged accesses.
    - If CONTROL.UBTI_EN is set to 1, then Branch Target Identification enabled for unprivileged accesses.

Note

You do not normally need to access the CONTROL.FPCA and CONTROL.SFPA bits directly. They are either set or cleared indirectly by the processor itself. For example, CONTROL.FPCA bit gets set to 1 when a floating-point instruction is executed for the first time in a program

### 5.2.3.1 Changing privilege level using the CONTROL register

The following code shows a simple sequence to change the privilege level by programming the CONTROL register:

```
MRS R0, CONTROL      ; Read CONTROL register into R0
ORR R0, R0, #1       ; Modify the R0 value to set the nPRIV field to switch to
                     ; unprivileged level on programming CONTROL register with R0
 value
MSR CONTROL, R0      ; CONTROL = R0, write CONTROL register with R0 contents
ISB                  ; Instruction Synchronization Barrier ensures the effect
                     ; of programing the CONTROL register applies
```

```
                   ; to instructions following ISB.
```

In C programming, CMSIS-CORE provides functions to read and write to the CONTROL register::

| CMSIS-CORE function name | Usage |
|---|---|
| `__get_CONTROL(void)` | Read the CONTROL register |
| `__set_CONTROL(uint32_t control)` | Write to the CONTROL register |

Note

Unlike the other special-purpose registers, the CONTROL register can be read in unprivileged level. This allows software to determine whether the current execution level is privileged or not.

You can detect if the current execution level is privileged by checking the value of IPSR and CONTROL registers:

```
int in_privileged(void)
{
    if (__get_IPSR() != 0) return 1; // TRUE when in handler mode
    else // In Thread mode
       if ((__get_CONTROL() & 0x1) == 0) return 1; // TRUE when nPRIV == 0
          else return 0; FALSE when nPRIV ==1
}
```

### 5.2.3.2 Stack pointer selection using the CONTROL register

The following code shows a how to set thread mode to use SP_Process as the current stack pointer:

```
MRS   R0, CONTROL  ; Read current CONTROL
ORRS  R0, R0, #0x2 ; Set SPSEL
MSR   CONTROL, R0  ; Write to CONTROL
ISB                ; Instruction Synchronization Barrier ensures
                   ; the effect of programing the CONTROL register applies
                   ; to instructions following ISB.
```

### 5.2.3.3 Stack pointer limit registers

Cortex-M processors use a fully descending stack operation model. This means that the stack pointers are decremented when more data is added to the stack. When too much data is pushed into the stack and the space consumed is more than the allocated stack space, overflowing stack data can corrupt the OS kernel data and memories used by other application tasks. This can cause various types of errors and may even result in security vulnerabilities.

The Armv8-M architecture includes stack limit registers to detect stack overflow errors. The stack limit registers are 32-bit and each stack pointer MSP and PSP has a corresponding stack limit register for stack limit checking:

- MSPLIM, Main Stack Pointer Limit Register
- PSPLIM, Process Stack Pointer Limit Register

By default, stack limit registers are reset to 0 (the lowest address in memory map) so that the stack limits will not be reached. Effectively this means that the stack limit checks are disabled at startup. The stack limit registers can be programmed when the processor is executing in privileged state. The stack limit registers (MSPLIM and PSPLIM) are accessed by `MRS` and `MSR` instructions similar to `MSP` and `PSP`. A stack can descend to a point until its stack limit value set in the stack limit register. Any attempt to descend further than its stack limit value causes a stack limit violation. If a stack limit violation occurs, a UsageFault or HardFaultfault exception is triggered.

Note

Writes to the lower 3 bits (bit 2 to bit 0) of the stack limit registers are ignored. Therefore stack limits are always aligned to doubleword boundaries.

The stack limit registers are banked when the Security Extension is implemented. The _S and _NS suffixes identify whether the stack limit register is for the Secure state or Non-secure state.

The following table shows the stack limit registers banked between Secure and Non-secure states.

| Stack limit register | Secure stack | Non-secure stack |
|---|---|---|
| Main stack pointer limit register | MSPLIM_S | MSPLIM_NS |
| Process stack pointer limit register | PSPLIM_S | PSPLIM_NS |

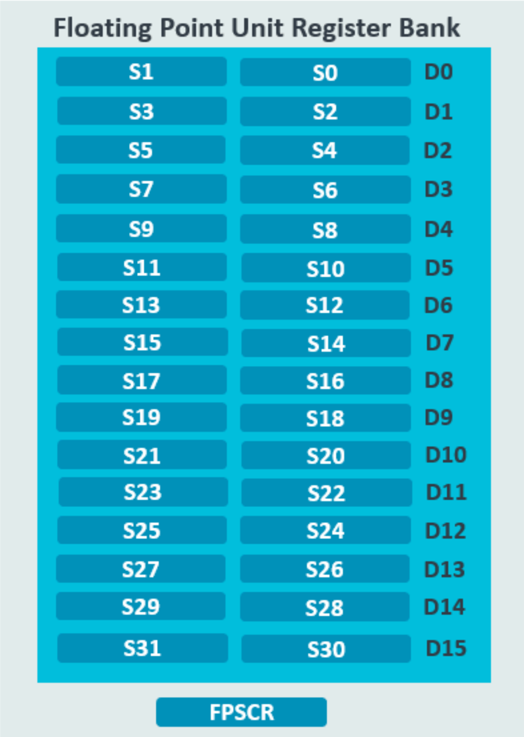
Similar to stack pointer access, the CMSIS-CORE software framework provides functions for accessing stack limit registers. The following table summarizes the stack limit register functions:

| CMIS-CORE function name | Usage details |
|---|---|
| `__get_MSPLIM(void)` | Gets the value of the MSPLIM register |
| `__get_PSPLIM(void)` | Gets the value of the PSPLIM register |
| `__set_MSPLIM(uint32_t limitofstack)` | Sets the value of the MSPLIM register |
| `__set_PSPLIM(unit32_t limitofstack)` | Sets the value of the PSPLIM register |

## 5.3 Floating-point registers

The Floating-Point Unit (FPU) hardware is an optional component in Armv8-M Mainline processors, for example the Cortex-M4, Cortex-M7, and Cortex-M33. If the FPU is available, it includes an additional register bank containing 32 registers (S0-S31) and a Floating-Point Status and Control Register (FPSCR). This is shown in the following diagram:

**Figure 5-8: FPU Registers**



Floating Point Unit Register Bank

| | | |
|---|---|---|
| S1 | S0 | D0 |
| S3 | S2 | D1 |
| S5 | S4 | D2 |
| S7 | S6 | D3 |
| S9 | S8 | D4 |
| S11 | S10 | D5 |
| S13 | S12 | D6 |
| S15 | S14 | D7 |
| S17 | S16 | D8 |
| S19 | S18 | D9 |
| S21 | S20 | D10 |
| S23 | S22 | D11 |
| S25 | S24 | D12 |
| S27 | S26 | D13 |
| S29 | S28 | D14 |
| S31 | S30 | D15 |

FPSCR

Each of the 32 bit registers S0-S31 ("S" stands for Single precision) can be accesseed individually using floating-point instructions, or accessed as a pair, using the register names D0-D15 ("D" stands for Double precision). For example, S1 and S0 are paired together to become D0, and S3 and S2 are paired together to become D1. The FPSCR register can only be accessed in privileged state. It contains various bit fields defining some of the floating-point operation behaviors and providing status information about floating-point operation results. By default, the FPU is disabled when the processor is out of reset to reduce power consumption. Hence before using floating-point operations, the FPU should be enabled by programming the Coprocessor Access Control Register (CPACR).

## 5.3.1 Using Floating-point extension

To use floating-point unit implemented within the processor, it needs to be enabled. The global enable of Floating-point is performed using `SCB->CPACR` register. This is handled inside

`SystemInit()` function. The FPU enabling code is enabled by C macros using the following preprocessing directives/macros in CMSIS-CORE.

| Preprocessing Directive | Description |
|---|---|
| __FPU_PRESENT | Indicates whether Cortex-M processor has an FPU. If it does, this macro is set to 1 by device specific header. |
| __FPU_USED | Indicates whether an FPU is being used. Must be set to 0 if __FPU_PRESENT is 0. It can either be 0 or 1 if __FPU_PRESENT is 1. This is set by compilation tools in their project settings. |
| __FPU_DP | Indicates whether the FPU support double precision operations. |

### 5.3.1.1 Compiler command line options

By selecting the FPU to be used in the project settings (within the IDE), the toolchain automatically sets the compiler options to include the FPU support.

Here are few example command line options that be used for Cortex-M processors like Cortex-M33:

For users of Arm Compiler 6 (which comes with Arm DS or DS-5), the following command line option can be used to enable FPU feature during compilation.

```
"armclang --target=arm-arm-none-eabi -marmv8-m.main -mfpu=fpv5-sp-d16 -mfloat-abi=hard"
```

For GNU C compiler (gcc), following command line can be used to enable FPU feature during compilation.

```
"arm-none-eabi-gcc -mthumb -march=armv8-m.main -mfpu=fpv5-sp-d16 -mfloat-abi=hard
```

### 5.3.1.2 ABI options

Application Binary Interface (ABI) refers to the specification that defines how the parameters and the results of the floating-point calculations are transferred across function boundaries. For example, even if you have an FPU in the processor, you may require to use C runtime library functions because many of the math functions require a sequence of calculations.

The ABI options affect:

- Whether the floating-point unit is used
- How parameters and results are passed between caller and callee functions.

There are three major ABI options available for floating-point. The options and its operation details are listed in the table below

| Arm C compiler 6 and GCC Floating-point ABI options | Description |
|---|---|
| -mfloat-abi=soft | Soft ABI without FPU hardare: All floating-point operations are handled by the runtime library functions. Values are passed through integer register bank |
| -mfloat-abi=softfp | Soft ABI with FPU hardware: This allows the compiled code to generate codes that directly access the FPU. But, if a calculation needs to use a runtime library function, a soft-float calling convention is used. Values are passed through integer register bank |
| -mfloat-abi=hard | Hard ABI: This allows the compiled code to generate codes that directly accesss the FPU and use FPU-specific calling conventions when calling runtime library functions |

### 5.3.2 Floating Point exceptions

Armv7-M and Armv8-M Mainline processors support the option of a hardware floating point unit (FPU). This is compliant with the IEEE 754 standard. The IEEE specification defines a number of invalid or out-of-range calculations and results, that it describes as “floating point exceptions”. The FPSCR provides six sticky bits so that sotware can check the sticky bit values to determine whether the calculations carried out were successful. Refer below table for additional information.

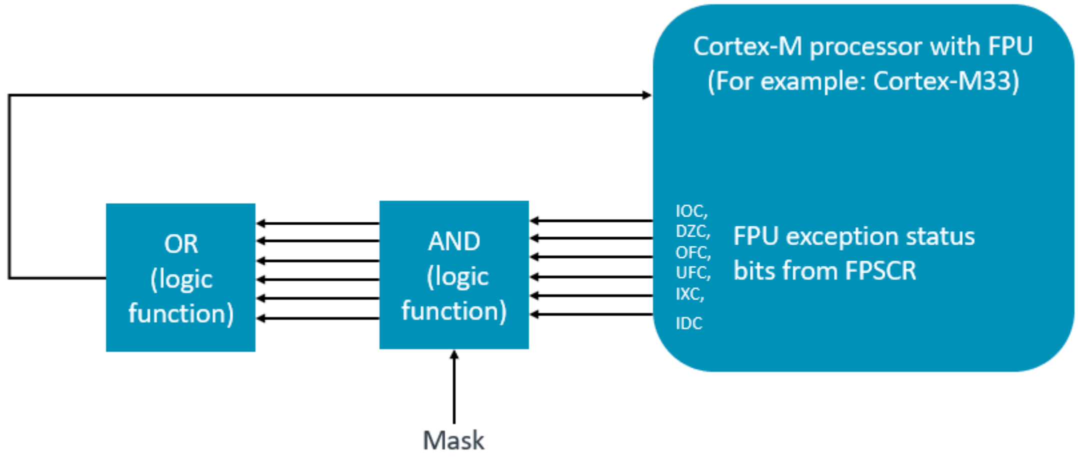
| Floating-point Exception | FPSCR bit | Example |
|---|---|---|
| Invalid Operation | IOC | Square root of a negative number (returns a QNaN by default) |
| Division by zero | DZC | Divide by zero or log(0) (returns +/- infinity by default) |
| Overflow | OFC | A result that is too large (returns +/- infinity by default) |
| Underflow | UFC | A result that is very small (returns a denormalized value) |
| Inexact | IXC | The result has been rounded (returns a rounded result by default) |
| Input Denormal | IDC | A denormalized input value is replaced with a zero in the calculation due to the Flush-to-zero mode. |

These Floating point “exceptions” should not be confused with the concept of exceptions in the Arm architecture. Refer Chapter-4 of Armv8-M Exception Model User Guide to understand the concept of exception with Floating-point context handling in Arm architecture. Floating point exceptions do not directly cause a change in program flow, but simply cause sticky flags in a status register to be set.

If you are designing software with high safety requirements, you could add the checking of the FPSCR in your code. However, in some instances not all floating-point calculations are carried out by the FPU. Some could be carried out by C runtime library function. If you are using C99, then you can examine and change the configuration of floating-point runtime library using `int fegetenv(envp);` and `int fesetenv(envp);`.

However, Cortex-M processors that support the optional FPU also present these internal register bits as output signal ports, and the system designer can choose whether to physically connect these signals individually, or combined together, to interrupt inputs on the processor. If the system designer connects the floating point exception outputs to general interrupt lines, then the corresponding interrupt will be invoked when a floating point exception occurs, and software will need to provide an exception handler for that interrupt to deal with the floating point condition.

**Figure 5-9: Floating point exception status bit for hardware exception generation**



By connecting the FPU exception status bits to the NVIC as shown in the Figure above, a system can trigger an interrupt when an error condition such as “divide by zero” or “overflow” occurs. Note that since interrupt events are imprecise, the generated exception could be delayed by a few cycles. This delay occurs even when the exception is not blocked by other exceptions. As a result, you will not be able to determine which floating point instruction triggered the exception. When the FPU exception status is used to trigger exceptions in NVIC, before the end of interrupt service routine, the exception handler needs to clear exception status bits in FPSCR register and stacked FPSCR. If exception status is not cleared, then the exception could get triggered again.

## 5.4 Memory-mapped registers

A number of system components such as Nested Vectored Interrupt Controller (NVIC), Memory Protection Unit (MPU), debug control registers, and coprocessor control registers, are memory-mapped in the Armv8-M architecture. These memory-mapped registers reside in an architecturally-defined fixed memory address space.

Since the majority of the system registers are memory-mapped registers, they are accessible using the load and store instructions or using pointers in C programming. A few simple examples are shared in this section, but you can find more complex examples in dedicated guides such as the Armv8-M Memory Model and MPU user guide, and the Armv8-M Security Extension user guide.

### 5.4.1 Example 1 - Enable IRQ0

The following is an example code sequence to enable an IRQ0 (IRQ0 -> Interrupt number #16).

```
; To enable an IRQ0, bit[0] of NVIC_ISER (NVIC Interrupt Set Enable Register) should
 be set to 1.
; This can be done by writing 0x1 to NVIC_ISER.
```

```
LDR R0,=0xE000E100    ; Load address of NVIC_ISER
MOV R1,#0x1          ; Bit[0] of R1 is set to 1
STR R1,[R0]          ; Write 1 to bit[0] of NVIC_ISER to enable IRQ0
DSB
```

If you are using a high-level language like C, then you can use the following CMSIS function:

```
void NVIC_EnableIRQ (IRQn_Type IRQn)
```

In this example, since IRQ0 needs to be enabled, the CMSIS function can be called as `NVIC_EnableIRQ(0)`.

### 5.4.2 Example 2 - Enable the Floating-Point Unit (FPU)

If the Floating-point (FP) Extension is implemented in a processor, then before using FP instructions in a program, the FP enable bits in Coprocessor Access Control Register (CPACR) should be set to 1.

The following is an example code sequence to enable the FPU:

```
; To enable the Floating-point Extension, bits[23:20] of the CPACR register
; should be programmed for CP10 and CP11 coprocessor register access

LDR R0,=0xE000ED88       ; Load address of CPACR
LDR R1,[R0]             ; Load the current value of CPACR
ORR R1, R1, #(0xF<<20)   ; Configure bits[23:20] for CP10 and CP11 coprocessor
 register access
STR R1,[R0]             ; Write modified CPACR value
DSB
ISB
```

If you are a using high-level language, you can use the CMSIS software framework. Besides an NVIC data structure in CMSIS-CORE, the System Control Block (SCB) data structure also contains some registers that can be used by software. So for this example, you can us the following code to enable the FPU.

```
SCB->CPACR|= 0x00F00000; // Enable the Floating-Point Unit for full access
```

# 6. References

Here are some resources related to material in this guide:

- Armv8-M Architecture Reference Manual
- Books:
    - The Definitive Guide to Arm Cortex-M3 and Cortex-M4 Processors - Joseph Yiu
    - The Definitive Guide to Arm Cortex-M23 and Cortex-M33 Processors - Joseph Yiu
- Cortex-M resources
- Procedure Call Standard for the Arm Architecture

# 7. Next steps

Refer to the following guides for more details about specific architectural extensions:

- Armv8-M Exception Model User Guide
- Armv8-M Memory Model and MPU User Guide
- Armv8-M Security Extension User Guide