Armv8-M Exception Model User Guide

Version 1.0

Non-Confidential

Issue 01

107706_0100_01_en

Armv8-M Exception Model User Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	19 July 2023	Non-Confidential	First release

Proprietary Notice

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

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

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

Inclusive language commitment

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

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

Contents

1. Introduction to the Armv8-M exception Model

This guide describes the Armv8-M exception model implemented in Cortex-M processors. This guide also provides examples to help explain the concepts it introduces.

The Arm architecture is divided into a number of profiles, targeting the requirements of different market segments. The Microcontroller profile, or "M-profile", architecture aims to provide the following characteristics:

- simplicity, to minimize costs for a given level of performance in the cost-sensitive target market of microcontrollers
- ease of use, to empower a very wide end-user base to program devices for diverse use cases without the need for deep expertise in Arm architecture and Arm assembly code
- efficient handling of many interrupt sources, as is typical of microcontroller products that often contain many sensors, timers, communication devices and other peripherals

Exceptions are conditions that cause a change in execution flow outside of the normal program flow defined by branch instructions in the program code. Exceptions include the following:

- general purpose interrupts that are typically triggered by devices sending an interrupt request signal to the processor
- system-defined interrupts from internal or external sources
- fault conditions from software or hardware
- system exceptions that can be used, for example, by Real-Time Operating Systems

The M-profile architecture has a very different exception handling model from the one used in legacy Arm architectures and in other Arm architecture profiles like Armv8-A, and Armv8-R.

The M-profile architecture uses a single execution mode, Handler mode, and a single stack pointer, the Main Stack Pointer, for all exception handling. User application code typically runs in Thread mode. Thread mode is programmable to run as privileged or unprivileged, and to share the Main Stack Pointer or use the alternate Process Stack Pointer. Privileged modes have access to all system resources while unprivileged Thread mode is restricted from accessing most of the processor's configuration status settings directly. Running user applications in unprivileged Thread mode using a separate stack located in memory that is private to that task, accessed via the Process Stack Pointer, limits the scope of erroneous or malicious code to affect other applications that are running, or the Operating System itself. Handler mode is always privileged, and is typically where Operating System functions run.

In the M-profile architecture, an interrupt controller called the Nested Vectored Interrupt Controller (NVIC), is built into the processor.

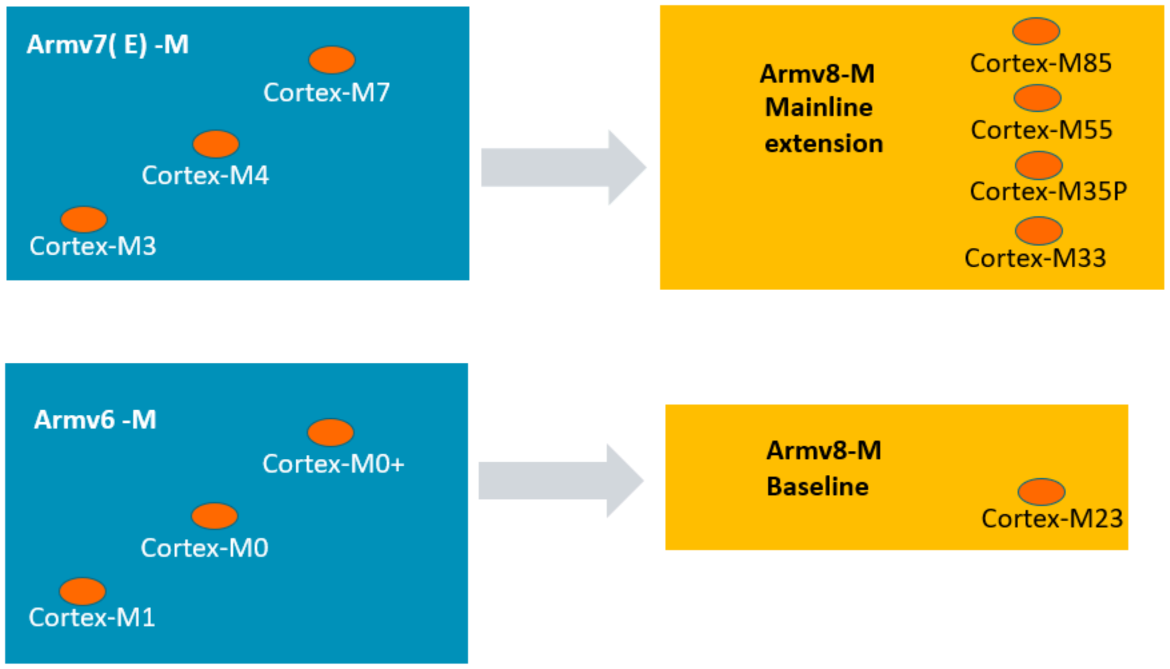
Nested means there is a built-in priority scheme, so that a handler for an interrupt or other exception at one priority level can be interrupted in favor of another exception at a higher priority level.

Vectored means that the NVIC knows the entry point address of the individual handler routine for each different exception, and uses that vector to take execution directly to the correct handler code routine. These vector values are stored in an area of memory known as the vector table, reserved for this purpose.

1.1 Compatibility between Armv6-M, Armv7-M, and Armv8-M architectures

The Armv8-M architecture is a successor to its predecessors, the Armv6-M and Armv7-M architectures. The following diagram shows the architectures that different Cortex-M processor products are built using:

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



Previously, there were two architecture versions for Cortex-M processors:

- The Armv6-M architecture, designed for ultra-low-power applications. This architecture supports a small and compact instruction set and is suitable for general data processing and I/O control tasks.
- The Armv7-M architecture, designed for mid-range and high-performance systems. This architecture supports richer instruction set including an optional DSP and Floating-point extensions.

The Armv8-M maintains a similar partitioning by splitting the architecture into two subprofiles:

- The Armv8-M Baseline architecture, designed for ultra-low-power designs. Its instruction set and features are a superset of the Armv6-M architecture.

- The Armv8-M Mainline architecture, designed for mainstream and high-performance designs. Its instruction set and features are a superset of the Armv7-M architecture.

From an architectural point of view, the Armv8-M Mainline architecture is an extension of the Armv8-M Baseline architecture. There are other architecture extensions including, for example:

- Digital Signal Processing (DSP) Extension
- Floating-point (FP) Extension
- Security Extension, TrustZone for Armv8-M
- M-profile Vector Extension (MVE) and so on.

Regarding the exception model specifically, upwards compatibility is retained in the following paths:

- Armv7-M -> Armv8-M Mainline
- Armv6-M -> Armv8-M Baseline

More details about exception model controls are described in later chapters of this guide.

2. Exceptions and interrupts overview

Interrupts are events typically generated by hardware, for example external peripherals or external pins, that cause changes in program flow control outside of the normal, programmed sequence.

When hardware or a peripheral needs service from the processor, the following sequence of events typically occur:

- The peripheral asserts an interrupt request to the processor.
- The processor suspends the currently executing task.
- The processor executes the Interrupt Service Routine (ISR) to service the peripheral.
- Once the ISR is complete, the processor resumes the suspended task.

Cortex-M processors provide a Nested Vectored Interrupt Controller (NVIC) for interrupt handling.

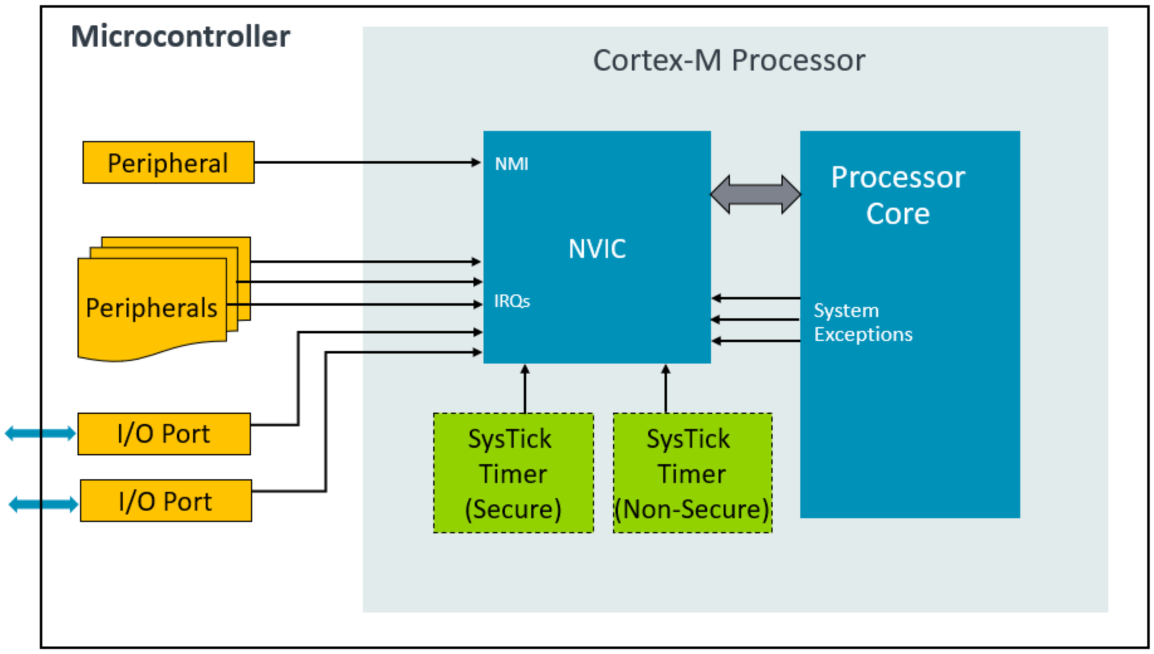
In addition to interrupts, there are other events that need servicing. These are called exceptions. In Arm terminology, an interrupt is a specific type of exception. Other exceptions in Cortex-M processors include:

- Faults
- System exceptions to support OS operations, including SVC, SysTick, and PendSV.

This chapter describes how the NVIC deals with exception requests from various sources.

In the Armv8-M architecture, the NVIC supports up to 496 general-purpose interrupt lines. However, processor implementations may limit the maximum to a lower number, often 240 or 480. Cortex-M processors let system designers choose any number of general-purpose interrupt lines appropriate to their system, up to the specified limit. In Cortex-M processors that include the Mainline extension, system designers can choose the number of bits implemented in the programmable priority value for each exception.

Figure 2-1: Sources of NVIC exceptions and interrupts in a Cortex-M based microcontroller



The NVIC is responsible for deciding which code stream the processor should be executing at any given time. It is also responsible for managing the current execution priority and the priorities assigned to all exception types. For most exceptions, the NVIC also takes care of whether each individual exception is enabled or not. Some exceptions are always enabled.

2.1 Exception types

The Armv8-M architecture supports several different types of system exception and external interrupt. Each exception type has a number, as follows:

- System exceptions: 1-15
- External interrupts: 16 and above

Most of the exceptions, including all external interrupts, have programmable priorities. However, several system exceptions have fixed priorities. The following table list all the system exceptions and interrupts supported by the Armv8-M architecture.

Exception number	Exception type	Priority	Description
1	Reset	-4 (highest)	Reset
2	NMI	-2	Non-Maskable Interrupt (NMI). This exception can be generated from external sources.
3	HardFault	-1 or -3	All fault conditions, if the corresponding fault handler is not enabled.

Exception number	Exception type	Priority	Description
4	MemManage Fault	Programmable	Memory Management Fault. This exception is caused by an MPU violation or by program execution from address locations with an eXecute Never (XN) attribute.
5	BusFault	Programmable	Bus error. This exception usually occurs when there is an error response from the bus responder. Since the error response can occur on both instruction fetch as well as data access, either type of access could result in a BusFault.
6	UsageFault	Programmable	Exceptions due to program error, for example, a divide-by-zero operation.
7	SecureFault	Programmable	Exceptions caused by security violations when the Security Extension is implemented in a system.
8-10	Reserved	NA	NA
11	SVC	Programmable	SuperVisor Call (SVC). This exception is normally used in an OS environment to allow application tasks to access system services.
12	DebugMonitor	Programmable	Debug monitor. This exception is triggered for debug events such as breakpoints and watchpoints.
13	Reserved	NA	-
14	PendSV	Programmable	Pendable Service Call. This exception is usually used in an OS environment for context switching operations.
15	SysTick	Programmable	System Tick Timer. An exception generated by a timer implemented within the processor.
16 - 495	Interrupt #0 - Interrupt #479	Programmable	Interrupts can be generated from on chip peripherals or from external sources.

The exception number is used to identify each exception. The value of the currently running exception is indicated by the Interrupt Program Status Register (ISPR). When creating applications which use device drivers that are CMSIS-CORE compliant, the interrupt identification is handled by an interrupt enumeration in the header file. CMSIS-CORE also defines the names of the system exception handlers. The enumeration definitions are used by various NVIC access functions in the CMSIS-CORE framework.

Exception number	Exception type	CMSIS-Core enumeration (IRQn)	Exception handler name in CMSIS-Core
1	Reset	-	Reset_Handler
2	NMI	NonMaskableInt_IRQn	NMI_Handler
3	HardFault	HardFault_IRQn	HardFault_Handler
4	MemManage Fault	MemoryManagement_IRQn	MemManage_Handler
5	BusFault	BusFault_IRQn	BusFault_Handler
6	UsageFault	UsageFault_IRQn	UsageFault_Handler
7	SecureFault	SecureFault_IRQnn	SecureFault_Handler
11	SVC	SVCall_IRQn	SVC_Handler
12	DebugMonitor	DebugMonitor_IRQn	DebugMon_Handler
14	PendSV	PendSV_IRQn	PendSV_Handler
15	SysTick	SysTick_IRQn	SysTick_Handler
16 - 495	Interrupt #0 - #479	Device-specific	Device-specific

After reset, all interrupts are disabled and given a priority level of 0. Before using any interrupt, you must do the following:

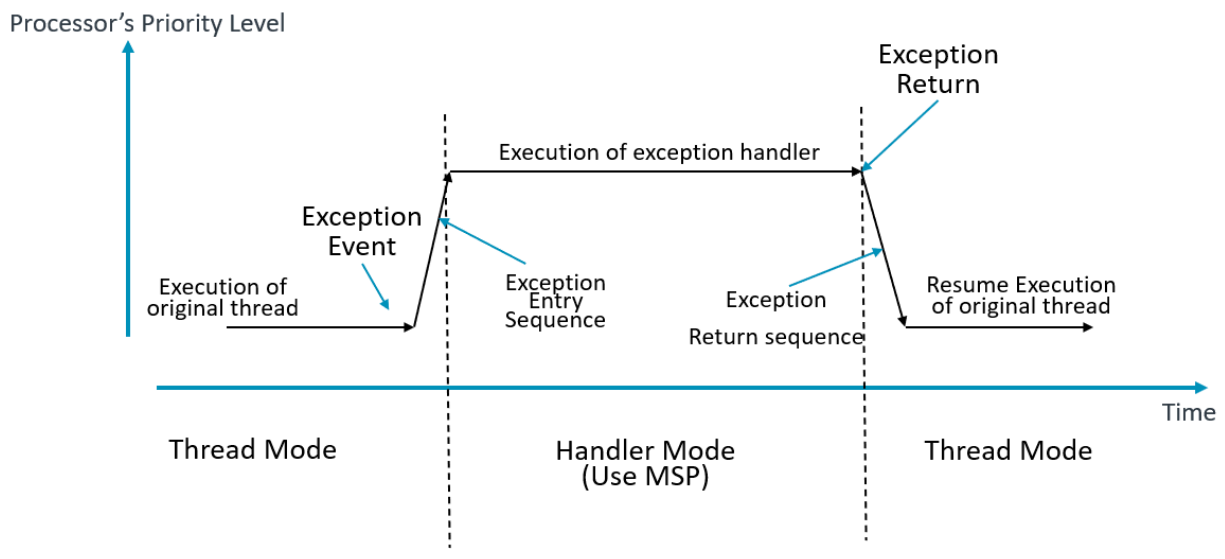
- Configure the priority level of the required interrupt.
- Enable the interrupt in NVIC.
- Provide a suitable interrupt service routine (ISR) to service the interrupt.
    - Ensure that the name of the ISR matches the name of the interrupt handler as defined in the vector table. This is required to enable the linker to place the starting address of the ISR in the vector table.
    - If you are not using CMSIS framework, then setup the entry in the vector table corresponding to the exception. See Section 2.4 for more details.

For most applications, these steps should be sufficient so that when the interrupt is triggered, the corresponding ISR is executed.

2.2 Exception handling sequences

Exception handling means allowing a code sequence that is currently running to be suspended so that the processor can run a different piece of code to deal with a situation that has been recognized at that moment. The following diagram shows a simplified view of how an exception might be handled:

Figure 2-2: Exception handling mechanism


Processor's Priority Level
Exception Return
Execution of exception handler
Exception Event
Execution of original thread
Exception Entry Sequence
Exception Return sequence
Resume Execution of original thread
Time
Thread Mode
Handler Mode (Use MSP)
Thread Mode

The following sections describe the operations performed in each of the steps shown in the diagram.

2.2.1 Acceptance of exception request

The processor accepts an exception request if the following conditions are met:

- An interrupt or exception event takes place, causing its pending state register to be set to 1.
- The processor is running and not in halted or in reset state.
- The exception is enabled. Note that NMI, HardFault, and SVC exceptions are always enabled.
- The exception has higher priority than current execution priority level.

2.2.2 Exception entry sequence

The exception entry sequence contains the following operations:

- Update the stack pointer.

  Depending on which stack is used, either the Main Stack Pointer (MSP) or Process Stack Pointer (PSP) value is adjusted immediately before the exception handler starts.
- Store register contents on the stack.

  The processor pushes the contents of a number of registers, including the return address, onto the stack. This process is called stacking, and it enables an exception handler to be written in a normal C function. If the processor was in Thread mode and was using the Process Stack Pointer (PSP), stacking uses the PSP. Otherwise, stacking uses the Main stack Pointer (MSP).
- Get the exception vector.

  The processor fetches the start address of the exception handler or ISR, and updates the Program Counter (PC) to this address.
- Fetch instructions.

  After the starting address of the exception handler is determined by reading the vector table, the instructions are then fetched.
- Update status.

  The processor updates the status in the NVIC and processor core registers. This includes the pending status and active status of the exception, and the PSR. The processor selects the Main Stack pointer (MSP) as the current stack pointer in the processor core.
- Update LR.

  The LR is updated with the special value EXC_RETURN. Branching to the EXC_RETURN value triggers the exception return sequence.

2.2.3 Exception handler execution

Within the exception handler, the event that triggered the exception request is serviced by software operations. The processor is in Handler mode when executing an exception handler. In Handler mode:

- The Main Stack Pointer (MSP) is used for any stack operations.
- The processor executes at a privileged access level.

If a higher priority exception arrives during the handler execution, then the new interrupt is accepted and the currently executing handler is suspended, preempted by the higher priority handler. This scenario is called a nested exception.

If another exception with the same or lower priority arrives during the handler execution, then the newly arrived exception remains in pending state until the current exception handler finishes.

When the exception handler finishes, the program code executes a return which causes the EXC_RETURN value to be loaded into Program Counter (PC). This triggers the exception return mechanism.

2.2.4 Exception return sequence

At the start of the handler, the EXC_RETURN value is placed in the Link Register. When an exception handler executes a branch to the EXC_RETURN value using one of the instructions in the table below, this causes an exception return. The active bit for the handler is automatically cleared at this point.

Return Instruction	Description
`BX <reg>`, or `BXNS <reg>`	If the LR contains the EXC_RETURN value when the exception handler ends, then the `BX LR` instruction can be used to perform exception return.
`POP {PC}`, or `POP {...,PC}`	In general, the LR value is pushed to the stack after entering the exception handler. In this situation, the `POP {PC}` or `{POP ...,PC}` (read stack along with other registers) instructions can be used as an exception return instruction.
Load (`LDR`) or Load Multiple (`LDM`)	Use `LDR` or `LDM` instructions with the PC as the destination register.

During exception return, the register values of the previously interrupted program that were saved to the stack during the exception entrance are automatically restored by the processor. This operation is called unstacking. When unstacking occurs, several NVIC registers and registers in the processor core are updated. Updated NVIC registers include the status register of interrupts. Updated registers in the processor core include the PSR, SP, and CONTROL.

The use of EXC_RETURN value for triggering exception returns allows exception handlers, including ISRs, to be written as a normal C function or subroutine.

2.3 Exception priority level definitions

Each exception in an Arm Cortex-M processor has an exception priority level. An exception pre-empts when its priority is higher than the current execution priority. In Cortex-M processors:

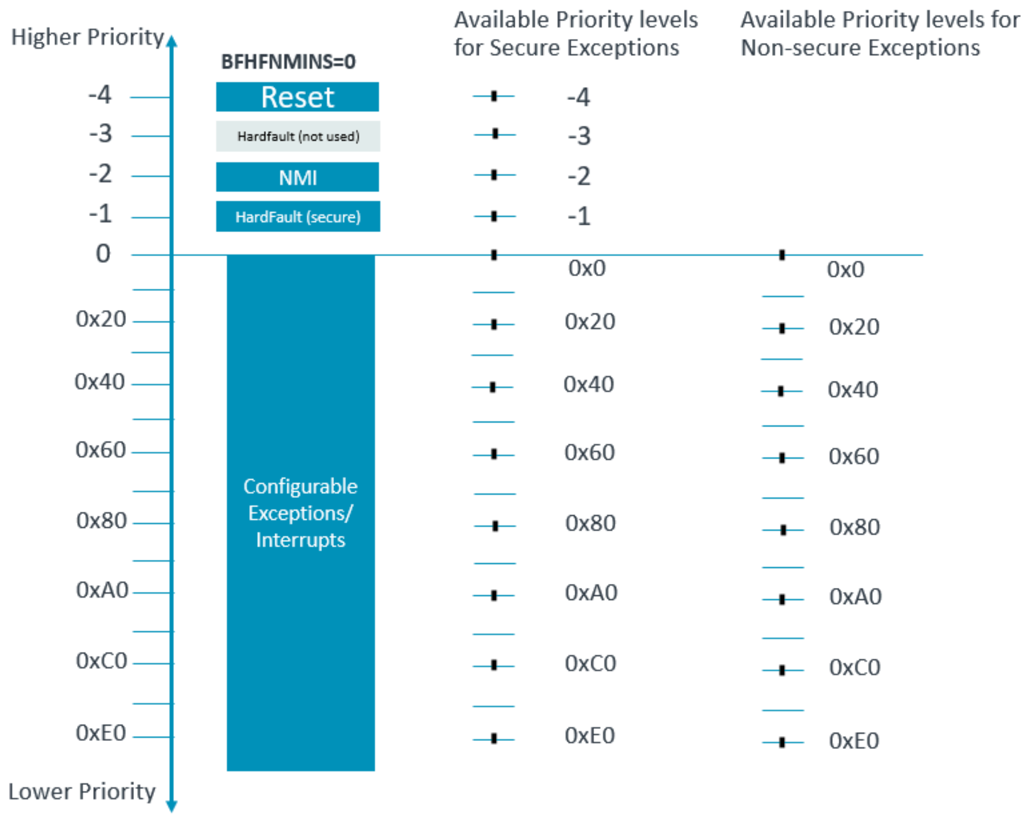
- A higher numerical value in the priority level register means a lower priority level, as shown in the diagram below.
- A priority level with a value of zero is the highest level for programmable exceptions.
- Some System exceptions, including NMI, HardFault, and Reset, have negative priority levels that are fixed. Therefore these system exceptions always have higher priorities than exceptions with programmable priority levels.

All general-purpose interrupts and most other exception types have programmable priorities. The priority value for each exception is located in an 8-bit field in a memory-mapped register.

For Armv6-M and Armv8-M Baseline, the programmable priority bits are defined to be the two most significant bits in the 8-bit field of the register.

For Armv7-M and Armv8-M Mainline, the number of implemented bits is configurable between three and eight. This is a hardware configuration chosen by the chip designer. Unimplemented bit positions are fixed at zero. The following diagram shows a simple example with a three bit priority level implementation.

Figure 2-3: Three bit Priority Level implementation



The exception priority level determines whether an incoming exception can be pre-empted by the processor:

- If the incoming exception event has a higher priority level than the processor's current priority level, then the exception request is accepted and the exception entry sequence starts.
- If the incoming exception event has the same or a lower priority level than the processor's current priority level, then the incoming exception request is held in pending state. This pending scenario can be caused by the following conditions:
    - The processor is already serving another exception of the same or higher priority level, or
    - A priority boosting register is set which changes the processor's current effective priority level to the same or higher priority level as the incoming exception.

The priority levels of exceptions and interrupts are controlled by priority level registers. These registers are memory-mapped and can be accessed only in privileged state. Programmable priorities for system exceptions are located in the System Handler Priority Registers.

Address	Name	Bits[31:24]	Bits[23:16]	Bits[15:8]	Bits[7:0]
`0xE000ED18`	SHPR1	SecureFault	UsageFault	BusFault	MemManage
`0xE000ED1C`	SHPR2	SVCall	Reserved	Reserved	Reserved
`0xE000ED20`	SHPR3	SysTick	PendSV	Reserved	DebugMonitor

The priority level for interrupts are controlled by the Interrupt Priority Registers (IPR).

Address	Name	Description
`0xE000E400` - `0xE000E5EF`	NVIC->IPR[0] to NVIC->IPR[495]	Defines the interrupt priority level for each external interrupt implemented in a system.

Note

Modifying the priority of an exception while it is in active state can cause undesirable side effects. Software should not modify the priority of an exception whilst it is in active state.

2.3.1 Priority grouping

The priority grouping feature allows exceptions with similar priorities to be grouped together. An exception will only pre-empt if it is in a higher priority group than the current execution priority. By increasing the size of the priority groups, software can reduce the maximum exception nesting depth, and therefore stack usage, but at the cost of increasing the latency of some exceptions.

For processors based on Armv7-M or Armv8-M architecture, the chip designer can define a hardware configuration so that all 8 bits are implemented in the priority level registers. However, instead of having the maximum of 256 pre-emption levels, the maximum number of pre-emption levels are limited to 128. This is because of the fact that 8-bit priority level registers are further divided into two halves:

- The upper half (left bits) is the group priority for the pre-emption control.

  The group priority level defines whether an interrupt can be handled when the processor is already running another interrupt handler.

- The lower half (right bits) is the subpriority.

  The subpriority level is only used when two exceptions with the same group priority level occur at the same time. In this situation, the exception with higher subpriority, that is the lower value, is handled first.

The dividing line between group priority and subpriority bits, called the binary point, is programmed by setting the AIRCR.PRIGROUP field. The PRIGROUP encoding is as follows:

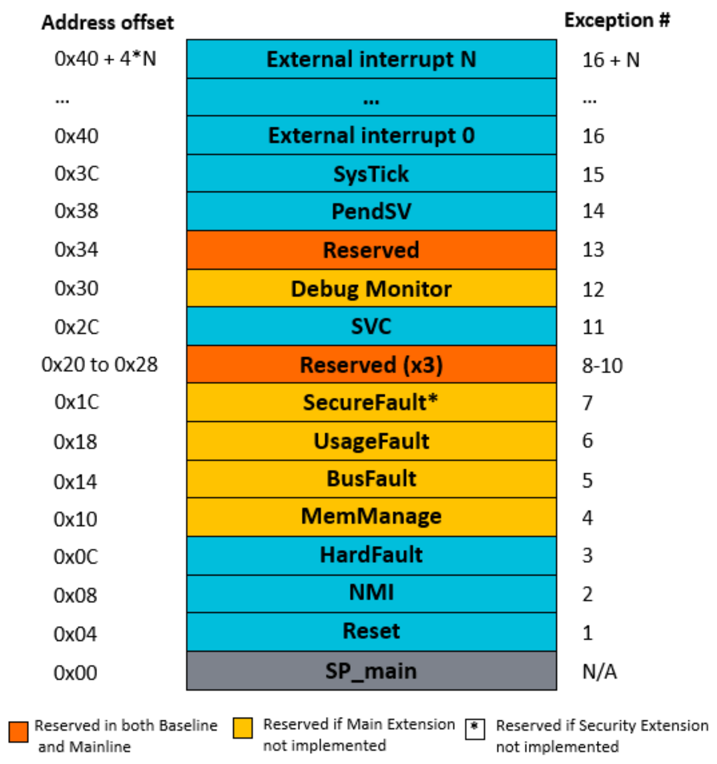
PRIGROUP	Group priority . Subpriority
`3'b000`	`ggggggg . s`
`3'b001`	`gggggg . ss`
`3'b010`	`ggggg . sss`
`3'b011`	`gggg . ssss`
`3'b100`	`ggg . sssss`
`3'b101`	`gg . ssssss`
`3'b110`	`g . sssssss`
`3'b111`	`. ssssssss`

There is no PRIGROUP encoding that results in all eight bits being treated as group priority. The maximum number of levels of nesting is limited to the fixed level exceptions HardFault and NMI, plus a number of programmable group priorities. Since PRIGROUP is architecturally limited to seven bits of group priority, the number of programmable priorities can never be more than 128.

2.4 Vector table

One of the most important steps of the exception entry sequence is to determine the starting address of the exception handler. In Cortex-M processors, this is automatically handled in the processor hardware by reading the address from the vector table. The vector table contains exception vectors, that is the starting address for each exception handler, arranged in the order of their exception numbers. Vector table entries are four bytes in size. The following diagram shows the layout of the vector table:

Figure 2-4: Vector table layout



When an exception is accepted by the processor, the starting address of the handler is read from the vector table. The address is calculated as follows:

```
Vector address = Exception_number * 4 + Vector_Table_Offset;
```

In a typical software project, the vector table is usually found in a device-specific file used by the start-up code. The first word in the vector table stores the initial value of the Main Stack Pointer (MSP). This value is copied into the MSP register during the reset sequence. This is needed because some exceptions, such as NMI, could occur immediately after the processor resets and before any other initialization steps.

Note

Bit zero of the exception handler entries in the vector table must be set to 1 to indicate that the T32 instruction set should be used, which is the only available instruction set on M-profile. Taking an exception to a vector entry with bit zero set to zero causes a UsageFault.

2.4.1 VTOR register and initialization

The Vector Table Offset Register (VTOR) is located at address `0xE000ED08`. This register specifies the location of the vector table in memory, and therefore where the processor should read the vector entry when taking an exception.

The M-profile architectures allow for various implementations of the VTOR. The reset value of VTOR is set by the device manufacturer. It is important to consult your device manufacturer's documentation to discover where the initial vector table must be placed.

You might need to relocate the vector table to another address. For example, an application might relocate the vector table from non-volatile memory to SRAM so that exception vectors can be configured at run-time. To relocate the vector table, do the following:

1. Copy the original vector table to the new location in memory.
2. Modify the exception vectors, if required.
3. Program the VTOR to point to the new vector table.
4. Execute a `DSB` instruction followed by an `ISB` instruction to ensure the change is effective immediately.

2.5 Exception states

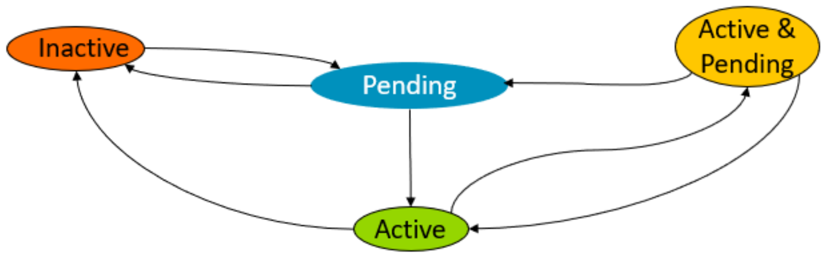
Each individual exception has its own state machine, and can be in one of four different states. The exception states are represented by pending and active bits in memory-mapped registers that are accessible to privileged software. Each exception has one pending and one active bit. Out of reset, all exceptions are in the inactive state. An exception remains in the inactive state until one of the following occurs:

1. The required condition occurs to trigger that exception. For example, an interrupt request signal is asserted to trigger that interrupt, or an error condition is signaled on a bus interface to trigger a BusFault exception.
2. Privileged software or a debugger writes to the register containing the pending bit for that exception to set that exception to the pending state.

3. In Armv8-M Mainline, privileged software or a debugger writes to the Software Triggered Interrupt Register (STIR) to set a particular interrupt to the pending state.

The second and third methods provide a convenient way to trigger exception from software in order to test exception handler code. Moving the exception directly to pending state in software removes the need to interact with external system components to trigger that exception. Once an exception has been triggered, it enters the pending state.

Figure 2-5: Exception states



An exception can leave the pending state in different ways. Normally, pending exceptions are handled in priority order. Immediately before the first instruction of the exception handler is executed, the exception transitions from the pending state to the active state. At this point, the pending bit is automatically cleared and the active bit is set.

The exception remains active until the final instruction of the handler has executed, the active bit is cleared, and the processor either returns to the suspended code stream or arbitrates to another pending exception that needs to be handled first according to its priority. Since an exception handler can be pre-empted by an exception with higher priority, it is possible, and common, for more than one exception to be active at the same time. In this case, the handler for the highest priority active exception runs, while the other active exceptions will have been pre-empted, with their execution states saved in exception stack frames.

The alternative way for an exception to leave the pending state is for privileged software to write directly to clear the pending bit in the relevant register. General interrupts each have one bit in the pending registers in the NVIC. Each individual pending interrupt can be cleared by writing a 1 to the corresponding bit position in the NVIC Interrupt Clear Pending address range, NVIC_ICPRn. System exceptions like NMI, SVCall, and SysTick can clear their pending status in the Interrupt Control and State Register (ICSR). The pending state of all other implemented system exceptions can be set or cleared in the SHCSR.

Note

Cortex-M processors support active HIGH external interrupt requests.

These external interrupt requests could be either of the following:

- Pulsed interrupt. A pulse must be at least one clock cycle long.
- Level triggered interrupt. A peripheral requesting the service asserts the request signal until it is cleared by a software operation inside the ISR. Software must

ensure that this request signal is cleared in the ISR at an early stage of the handler. If the interrupt is cleared towards the end of handler, it is possible that by the time it reaches the external peripheral, the processor could move ahead in the execution. That is, the level triggered interrupt could still be active HIGH and the processor could re-enter the interrupt handler once again.

2.6 Stack frames

As explained in Exception handling sequences, Cortex-M processors use stacking to automatically push a number of registers to stack memory on exception entry, then use unstacking to restore those registers from stack memory when returning to the pre-empted context.

For easier development of Cortex-M software, stacking and unstacking works in a way that enables most exception and interrupt handlers to be programmed as ordinary C functions, without the need for toolchain-specific keywords to specify that they are exception handlers. To understand how this is achieved, you need to understand how the C function interface behaves. This is defined in the Procedure Call Standard for Arm Architecture.

The Procedure Call Standard specification states that C functions can modify registers R0-R3, R12, LR (R14), and PSR without preserving their previous values. If a floating-point unit is present and enabled, registers S0-S15 and FPSCR can also be modified by C functions. The contents of the other registers can also be modified within C functions, but the contents of these other registers must be saved to the stack before being modified, then the original values restored before leaving the C function.

Based on the Procedural Call Standard requirements, the registers can be divided as follows:

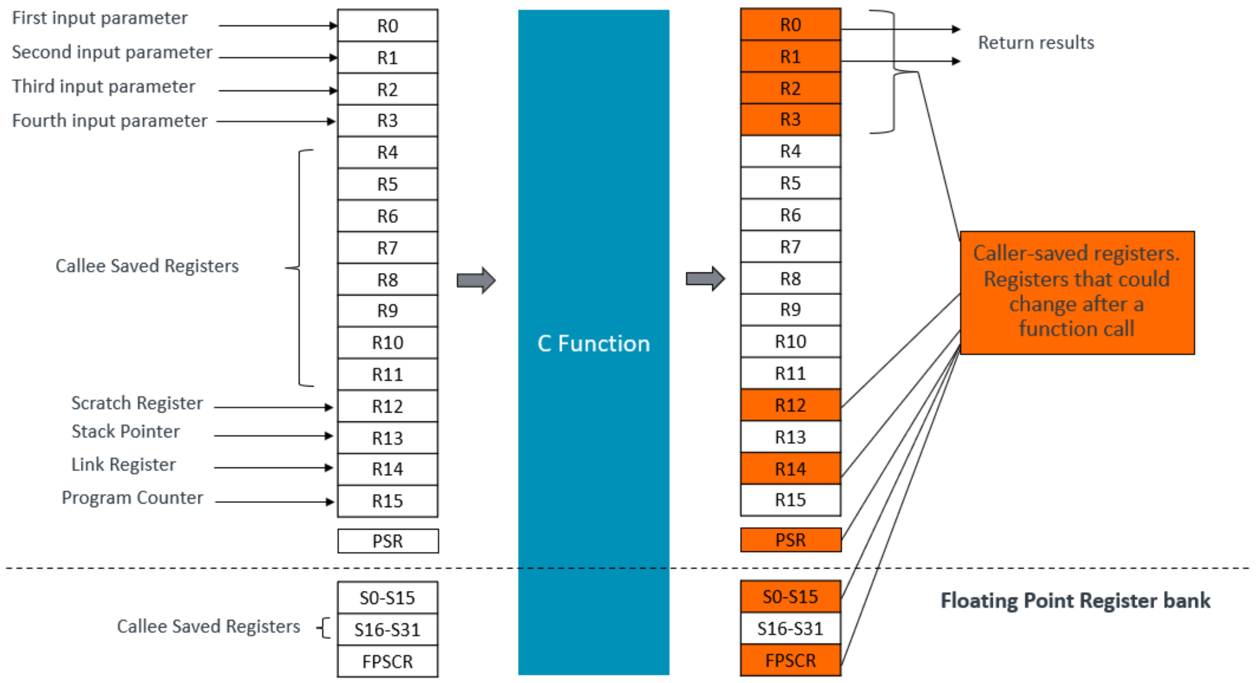
- Caller-saved registers: R0-R3, R12, LR, plus S0-S15 and FPSCR if a floating-point unit is implemented.

  If the data in these registers need to be used by C function call, the caller needs to save it before calling a C function.
- Callee-saved registers: R4-R11 plus S16-S31 if a floating-point unit is implemented.

  If a C function needs to modify any of these registers, the C function must first push the affected registers to the stack and then restore them before returning to caller code.

The figure below gives a pictorial representation of caller-saved and callee-saved registers.

Figure 2-6: Caller-saved registers and callee-saved registers



In addition to defining caller- and callee-saved register requirements, the Procedural Call Standard specification also specifies how parameters and results can be passed between caller and callee. In a simple scenario, registers R0-R3 can be used as input parameters for C functions. Another requirement of Procedural Call Standard is that value of stack pointer must be aligned to doubleword boundaries at a function interface. As part of the exception handling sequence, Cortex-M processor automatically aligns the stack pointer to a doubleword boundary.

Note

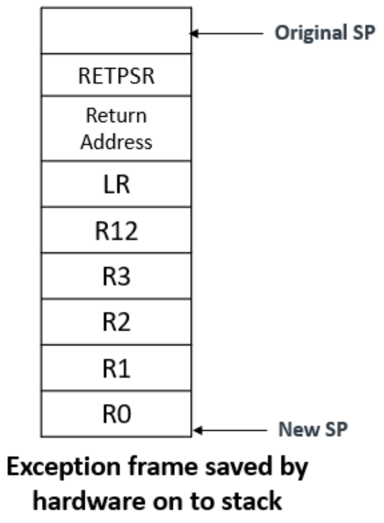
The automatic stacking and unstacking operations use the currently selected stack pointer.

The following simple example shows you can create an exception handler in C:

```
void Timer0_Handler(void)
{
   ... //
   ... // Clear timer interrupt request at the timer peripheral
}
```

The function name, `Timer0_Handler` in the above example, must match the handler name declared in the vector table used in the device-specific startup code. To allow a C function to be used as an exception handler, the exception mechanism needs to automatically save the caller-saved registers at exception entry and restore them at exception return. These operations are under the control of processor hardware. In this way, the registers contain the same values when returning to the pre-empted context as they had before the exception took place. The exception stacking operations place the caller-saved registers in a data block on the stack. This data block is called an exception stack frame. The Armv8-M architecture defines the layout of data inside the exception stack frame. The following diagram shows the exception stack frame layout when a floating-point unit is not present:

Figure 2-7: Basic exception stack frame



For interrupt handling, the automatic stacking and unstacking is handled by processor and is transparent to software. However, it is useful to understand the stack frame format in the following scenarios:

- When OS software developers need to create context switching codes or OS services through SVC exceptions. For example project source code, see svc-number-as-parameter.
- When debugging software after the processor enters a fault exception.

At a minimum, an exception stack frame must contain at least eight words, as shown in the diagram above. These eight words of data contain the caller-saved registers in the regular register bank and information to enable the pre-empted software to resume. Because exception handlers can be implemented as normal C functions, the contents of R0-R3, R12, LR, and RETPSR must be saved. Unlike function calls, the return address for exception handlers is not stored in the LR.

The stack frame can be complex when the floating-point extension is implemented and enabled. This is because the processor must save the caller-saved registers S0-S15 and FPSCR to the exception stack frame. There are two methods of saving floating-point caller-saved registers on to the exception stack frame:

- Automatic FP stacking
- Lazy FP context saving

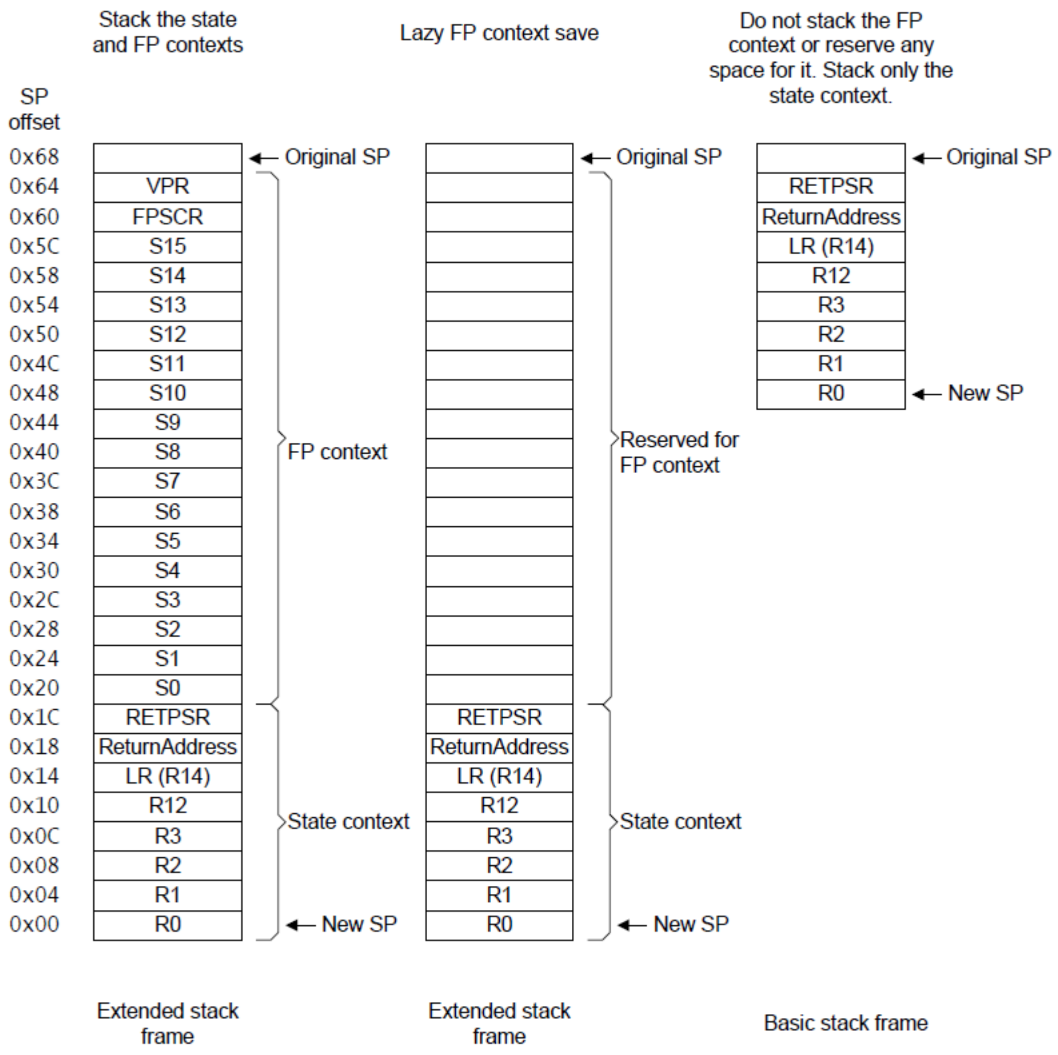
The following figure shows the stack frame layout for three different scenarios:

1. Basic stack frame with no floating-point extension.
2. Extended stack frame with lazy FP context save.

   Floating-point is enabled, and floating-pointer caller-saved registers are only saved to the stack frame on execution of a floating-point instruction inside the exception handler.
3. Extended stack frame with automatic stacking.

   On exception entry, the caller-saved registers in the floating-point register bank S0-S15 and FPSCR are saved to the exception stack frame.

Figure 2-8: Exception stack frame types



See Floating-point context handling mechanisms for more information about floating-point exception stack frames.

2.7 EXC_RETURN

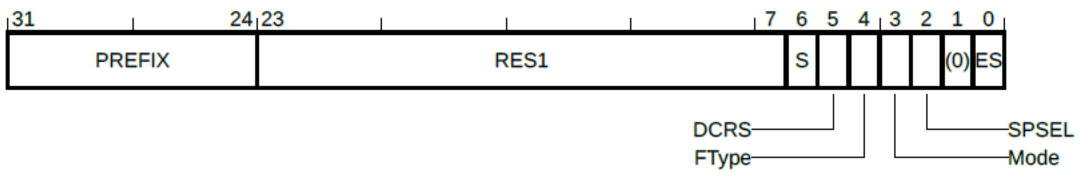
The Exception Return Payload value, EXC_RETURN, is automatically populated into the Link Register on entry to an exception. In C functions, function return is normally carried out by branching to the return address, which is stored in LR on a function call, for example by using the `BX LR` instruction. Branching to the EXC_RETURN value that was placed in LR at the start of an exception triggers the exception return operation. Because of this, any regular C function can be used as an exception handler.

The payload value includes the following:

- A prefix in the high-order bits that identifies it as an EXC_RETURN.
- Individual low-order bits that carry information to identify specific details of which context is being returned to. Some of these bits apply to specific extensions, and are reserved if those extensions are not configured.

The bitfields are defined as follows:

Figure 2-9: Bit fields of EXC_RETURN



Name	Position	Description
PREFIX	31:24	`0xFF`. This means that the payload represents an address in the Vendor_SYS region of the address space, which is architecturally a non-executable (XN) address, and can never be a valid branch target address. This triggers the exception return mechanism.
RES1	23:7	Padding, reserved with each bit set to one.
S	6	Indicates whether the exception stack frame is on a secure or a non-secure stack. One means secure. Always zero if the security extension is not configured. In Armv6-M and Armv7-M this bit position is always reserved as one.
DCRS	5	Default callee register stacking used. One when the default rules are used for callee register stacking, zero when callee register stacking is skipped because the callee registers are already stacked. This bit is always one if the security extension is not configured.
FType	4	Stack frame type. One for the standard stack frame, or zero for the extended floating-point stack frame format.
Mode	3	Indicates which mode was pre-empted. Zero for Handler mode or one for Thread mode.
SPSEL	2	Saves the value of CONTROL.SPSEL in the domain that is handling the exception. The handler is run with CONTROL.SPSEL cleared, to select the Main stack. The saved value is restored during exception return.
(0)	1	Reserved as zero.
ES	0	Indicates the security domain that is handling the exception. Zero for non-secure, one for secure. Always zero if the security extension is not configured. In Armv6-M and Armv7-M this bit position is used to indicate Thumb state execution and is always one.

2.8 Classification of synchronous and asynchronous exceptions

Exceptions that are attributed to the currently-executing instruction are said to be synchronous to code execution. In general, these exceptions must be dealt with at the point where they occur, and code execution cannot proceed until the exception has been handled. For this reason, synchronous exceptions are escalated to HardFault if they are not enabled or do not have sufficient priority to pre-empt the current context.

Synchronous exceptions of type UsageFault, BusFault (precise), and MemManage Fault populate bits in their respective Fault Status Registers UFSR, BFSR and MMFSR that together form the Configurable Fault Status Register (CFSR). These fault status registers give more detail about the exact cause of the fault. MemManage Faults and Synchronous BusFaults also populate their respective fault address registers, MMFAR and BFAR, to indicate the address that was accessed to cause the fault. SecureFault is also a Synchronous exception. SecureFault populates the exact cause of fault in a dedicated SecureFault Status Register (SFSR) which is not part of CFSR. Also, when a SecureFault occurs, wherever applicable the SecureFault Address Register (SFAR) is populated.

Exceptions that are not attributable to the currently executing instruction are asynchronous to code execution. These exceptions are set to the pending state. If they do not have sufficient priority to pre-empt, they are handled in due course when they are both enabled and are arbitrated to have the highest priority amongst the current execution context and all pending exceptions. Examples of this type of asynchronous exception include:

- External interrupts, including NMI
- SysTick interrupt
- PendSV exception
- Asynchronous BusFault

Note

Unlike other asynchronous exceptions, an asynchronous BusFault escalates to HardFault if the BusFault is disabled.

Exceptions can come from internal or external sources. Internal exceptions can often be attributed to the currently executing instruction, for example UsageFaults, MemManage Faults, and SVC. External exceptions are often unrelated to the currently executing instruction, for example Reset or the various interrupts.

2.9 NVIC registers for interrupt management

There are several registers in NVIC for interrupt control. These registers are located in the System Control Space (SCS) address range. The following table lists these registers:

Management function	Register type	Address	CMSIS-Core symbol	Notes
Enable interrupts	Interrupt Set Enable Registers	`0xE000E100` to `0xE000E13C`	NVIC->ISER[0] to NVIC->ISER[15]	Write 1 to set enable
Disable interrupts	Interrupt Clear Enable Registers	`0xE000E180` to `0xE000E1BC`	NVIC->ICER[0] to NVIC_ICER[15]	Write 1 to clear enable
Pend an interrupt	Interrupt Set Pending registers	`0xE000E200` to `0xE000E23C`	NVIC->ISPR[0] to NVIC->ISPR[15]	Write 1 to set pending status
Clear a pending interrupt	Interrupt Clear Pending registers	`0xE000E280` to `0xE000E2BC`	NVIC->ICPR[0] to NVIC->ICPR[15]	Write 1 to clear pending status
Get the interrupt status	Interrupt Active Bit registers	`0xE000E300` to `0xE000E33C`	NVIC->IABR[0] to NVIC->IABR[15]	Read only Interrupt Active status bits
Define the priority level of an interrupt	Interrupt Priority Registers	`0xE000E400` to `0xE000E5EF`	NVIC->IPR[0] to NVIC->IPR[495 or 123]	Interrupt Priority level for each external interrupt. In Armv8-M Mainline, each IPR register contains 256 priority level (8 bits). In Armv8-M Baseline, each IPR register contains 4 priority levels (2 bit)
Define the interrupt's target security state when the Security extension is implemented	Interrupt Target Non-secure State Registers	`0xE000E380` to `0xE000E3BC`	NVIC->ITNS[0] to NVIC->ITNS[15]	Write 1 to set an interrupt to Non-secure. Clear to 0 to set an interrupt to Secure

All these registers, with the exception of the Software Trigger Interrupt Register (STIR), are only accessible at privileged level. The STIR is accessible at privileged level only by default. However, it can be configured to be accessible at unprivileged level by setting the USERSETMPEND bit in Configuration Control Register (CCR).

Out of system reset, the initial status of the external interrupts are as follows:

- All interrupts are disabled.
- All interrupts have a priority level of 0, the highest programmable level.
- No interrupt is in pending or active state.

The Interrupt Enable register is programmed using two addresses. To set the enable bit, write to the NVIC->ISER[n] register address. To clear the enable bit, write to the NVIC->ICER[n] register address. The ISER and ICER registers are 32 bits wide, with each bit representing one interrupt input. If your implementation contains 32 or fewer interrupts, then you only have the NVIC->ISER[0] and NVIC->ICER[0] registers. However, if you have more than 32 interrupts, then bit 0 of NVIC->ISER[1] and bit 0 of NVIC->ICER[1] register are implemented.

If an interrupt takes place but cannot be executed immediately, for example because another higher-priority interrupt handler is running, then the interrupt is pended. The interrupt-pending status is accessible through the Interrupt Set Pending Register NVIC->ISPR[n]. The interrupt pending state can be cleared by the Interrupt Clear Pending Register NVIC->ICPR[n]. The pending status controls are similar to the interrupt enable registers. If there are more than 32 external interrupt inputs, more than one ISPR and ICPR register is implemented. Since the values of pending status registers can be changed by software, you can do the following:

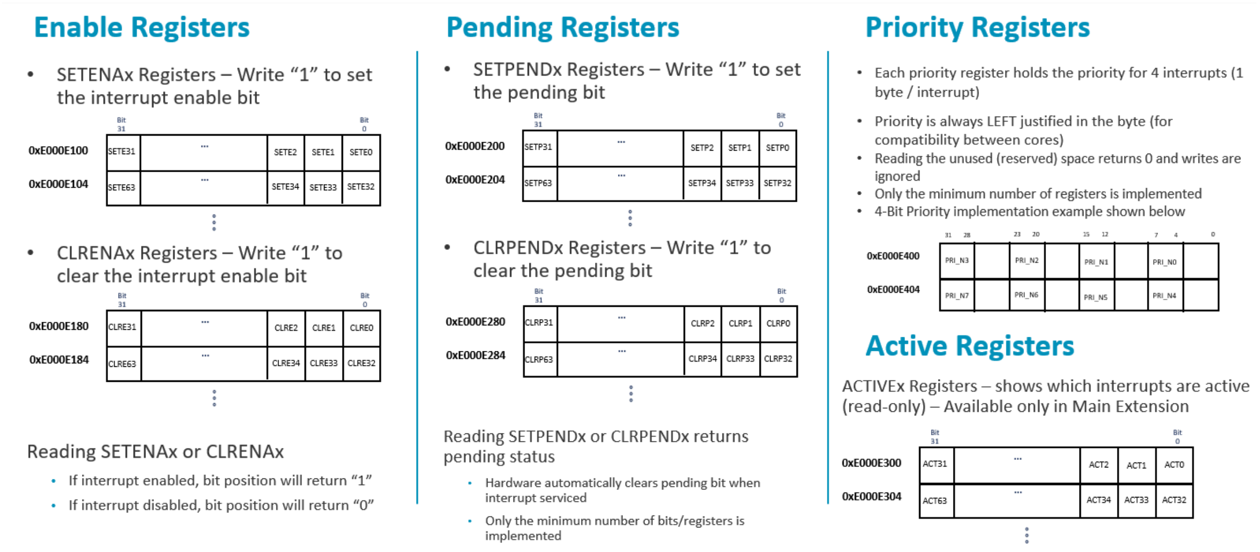
- Cancel a pended exception by a software writing to the NVIC->ICPR[n] register.
- Generate a software interrupt by a software writing to the NVIC->ISPR[n] register.

Each external interrupt has an active status bit. When the processor executes the interrupt handler, the corresponding active status bit is set to 1 and is cleared when the interrupt return is executed. However, during an Interrupt Service Routine (ISR) execution, another higher priority interrupt might occur and cause pre-emption. This results in a nested exception scenario. In this case, the previous interrupt is still defined as active. During this type of nested exception scenario, the IPSR register only shows the currently executing exception number. To identify the other interrupts which are in active state, use the Interrupt Active Status Register, NVIC->IABR[n].

Each interrupt has an associated priority level register, as described in Exception priority level definitions. The number of priority level registers depends on the number of external interrupts implemented in the system.

The following diagram summarizes the NVIC registers used for external interrupt configuration.

Figure 2-10: Summary of NVIC registers for external interrupts



Enable Registers

- SETENAx Registers – Write "1" to set the interrupt enable bit
- CLRENAx Registers – Write "1" to clear the interrupt enable bit

Reading SETENAx or CLRENAx

- If interrupt enabled, bit position will return "1"
- If interrupt disabled, bit position will return "0"

Pending Registers

- SETPENDx Registers – Write "1" to set the pending bit
- CLRPENDx Registers – Write "1" to clear the pending bit

Reading SETPENDx or CLRPENDx returns pending status

- Hardware automatically clears pending bit when interrupt serviced
- Only the minimum number of bits/registers is implemented

Priority Registers

- Each priority register holds the priority for 4 interrupts (1 byte / interrupt)
- Priority is always LEFT justified in the byte (for compatibility between cores)
- Reading the unused (reserved) space returns 0 and writes are ignored
- Only the minimum number of registers is implemented
- 4-Bit Priority implementation example shown below

Active Registers

ACTIVEx Registers – shows which interrupts are active (read-only) – Available only in Main Extension

To make it easier to manage interrupts and exceptions, the CMSIS-CORE header files provide a number of access functions to enable a portable software interface. For general application programming, best practice is to use the CMSIS-CORE access functions for interrupt management. The following table summarizes the most commonly used interrupt control functions.

Function	Usage
`void NVIC_EnableIRQ (IRQn_Type IRQn)`	Enables an external interrupt
`void NVIC_DisableIRQ (IRQn_Type IRQn)`	Disables an external interrupt
`void NVIC_SetPriority (IRQn_Type IRQn, uint32_t priority)`	Sets the priority of an interrupt
`void __enable_irq (void)`	Clears PRIMASK to enable interrupts
`void __disable_irq (void)`	Sets PRIMASK to disable interrupts
`uint32_t NVIC_GetPriority(IRQn_Type IRQn)`	Reads the priority level of an interrupt or configurable exception
`void NVIC_SetPendingIRQ (IRQn_Type IRQn)`	Sets the pending state of an interrupt
`void NVIC_ClearPendingIRQ (IRQn_Type IRQn)`	Clears the pending state of an interrupt
`uint32_t NVIC_GetPendingIRQ (IRQn_Type IRQn)`	Reads the pending status of an interrupt, returning 0 or 1
`uint32_t NVIC_GetActive (IRQn_Type IRQn)`	Reads the active state of an interrupt, returning 0 or 1
`uint32_t NVIC_SetTargetState (IRQn_Type IRQn)`	Configures the interrupt target state to Non-Secure and returns the interrupt's target state for checking. 0 = Secure, 1 = Non-secure.
`uint32_t NVIC_ClearTargetState (IRQn_Type IRQn)`	Configures the interrupt target state to Secure and returns the interrupt's target state for checking. 0 = Secure, 1 = Non-secure.
`uint32_t NVIC_GetTargetState (IRQn_Type IRQn)`	Reads the target security state of an interrupt. 0 = Secure, 1 = Non-secure.

To determine the implemented width of the interrupt priority level registers, or the number of priority levels available in the NVIC, you can use the `__NVIC_PRIO_BITS` C preprocessing macro in the CMSIS-CORE header file. Alternatively, you can write `0xFF` to one of the interrupt priority level registers and then read it back to check how many bits are set. For example, if your device contains only 8 priority levels (3 bits), then the read value will be `0xE0`.

2.10 SCB registers for system exception management

The System Control Block (SCB) contains registers for the following:

- System management, including system exceptions. See System exceptions for more information.
- Fault handling. See Fault exceptions and their causes for more information.
- Access management for the Coprocessor and Arm Custom Instructions features.
- A number of ID registers to determine the processor features.

The following table summarizes the SCB registers and the CMSIS-CORE standardized software interface for accessing these registers.

Address	Register	CMSIS-CORE symbol	Function
`0xE000ED04`	Interrupt Control and State Register	SCB->ICSR	Provides controls and status bit information for system exceptions.
`0xE000ED08`	Vector Table Offset Register	SCB->VTOR	Enables the vector table to be relocated to other address locations.
`0xE000ED0C`	Application Interrupt/Reset Control Register	SCB->AIRCR	Configures priority grouping and self-reset control.
`0xE000ED18` - `0xE000ED23`	System Handler Priority Registers	SCB->SHP[n]	Provides the priority level bit fields for system exceptions including SVC, PendSV, SysTick, and DebugMonitor exceptions.
`0xE000ED24`	System Handle Control and State Register	SCB->SHCSR	Provides the bit fields to enable or disable the configurable fault exceptions including BusFault, MemManage Fault, UsageFault, and SecureFault. It also contains bit fields to indicate the status, Active or Pending, of system exceptions.

Software can use the `NVIC_SetPriority()` and `NVIC_GetPriority()` functions to configure and access the priority levels of system exceptions. However, since CMSIS-CORE does not define specific APIs for system exception management tasks, software needs to directly access the SCB registers.

For example:

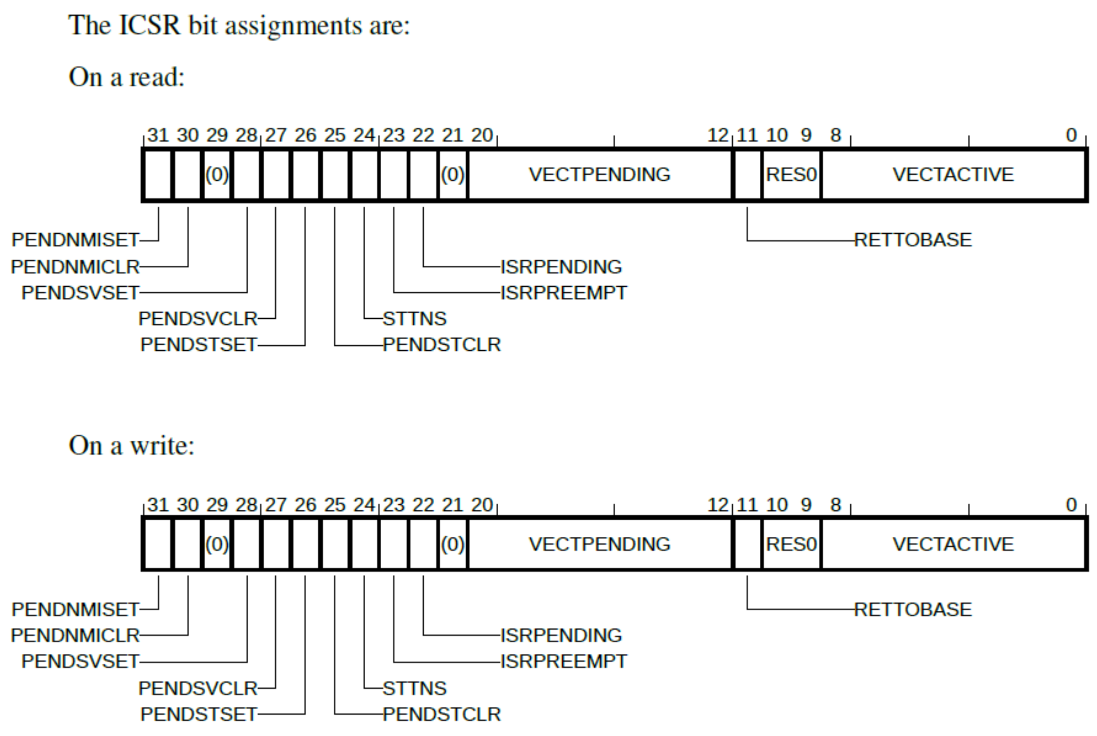
- To trigger PendSV, NMI, or SysTick exceptions, software must write to the ICSR register.
- To enable the configurable fault exceptions BusFault, UsageFault, MemManage Fault, and SecureFault, software must write to the SHCSR register.

2.10.1 Interrupt Control and State Register (SCB->ICSR)

The ICSR register is used by application code to do the following:

- Set and clear the pending status of system exceptions, including SysTick, PendSV, and NMI.
- Determine the currently executing exception number by reading VECTACTIVE bits. Reading the VECTACTIVE bit field in the ICSR register from an external debugger is equivalent to the IPSR, that can be easily read from the debugger.

Figure 2-11: Bit fields of ICSR


The ICSR bit assignments are:

On a read:

On a write:

Bits	Name	Description
31	NMIPENDSET	Write 1 to this bit to pend an NMI exception from software. The read value of this bit indicates the NMI's pending status.
30	NMIPENDCLR	Write 1 to clear the NMI Pending status.
28	PENDSVSET	Write 1 to pend a PendSV exception. The read value indicates the pending status of the PendSV exception.
27	PENDSVCLR	Write 1 to clear a PendSV pending status.
26	PENDSTSET	Write 1 to pend a SysTick exception. The read value indicates the pending status of the SysTick exception.
25	PENDSTCLR	Write 1 to clear the SysTick pending status.
24	STTNS	SysTick targets Non-Secure. Refer to the Security extension user guide for more details
23	ISRPREEMPT	This is a read-only bit that indicates that a pending interrupt is going to be active in the next step, when single-step debugging.
22	ISRPENDING	Interrupt Pending status.
20:12	VECTPENDING	Indicates the Pending ISR number.
11	RETTOBASE	Set to 1 when (a) the processor is running an exception handler and (b) there are no other pending exceptions. If this bit is 1 and when there is an interrupt return, the processor will return to Thread.
8:0	VECTACTIVE	This bit field indicates the exception number of current executing ISR.

For more information about the ICSR register bits, see Section D1 of the Armv8-M Architecture Reference Manual.

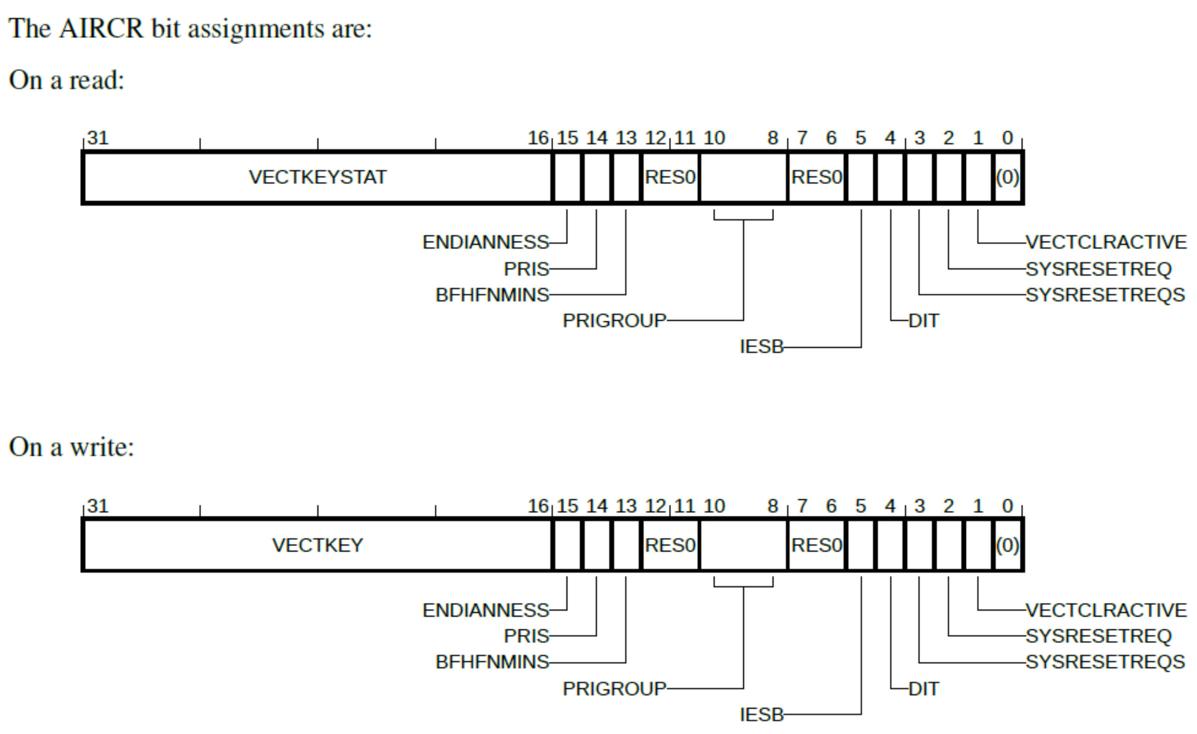
2.10.2 Application Interrupt and Reset Control Register (SCB->AIRCR)

The AIRCR register is used for the following:

- Controlling the priority grouping in exception priority management.
- Providing information about the endianness of the system.
- Providing a self-reset feature.

The priority grouping feature is described in Priority grouping. The PRIGROUP bit field can be accessed by the CMSIS-CORE functions `NVIC_SetPriorityGrouping()` and `NVIC_GetPriorityGrouping()`. SYSRESETREQ is used for software-generated reset. This type of reset is generally used by debugger to reset the hardware target.

Figure 2-12: Bit fields of AIRCR


The AIRCR bit assignments are:

On a read:

On a write:

The following table summarizes the AIRCR bit fields:

Bits	Name	Description
31:16	VECTKEY	Vector Key. The value `0x05FA` must be written to these bits when writing to AIRCR register, otherwise writes to the AIRCR register are ignored.
15	ENDIANNESS	This is a Read-only bit indicating the endianness for data. 1 indicates big-endian (BE-8) and 0 indicates little endian.
14	PRIS	This bit is used to prioritize Secure exceptions. Set this bit to deprioritize Non-secure exceptions. Refer to the Security Extension User Guide for more details.

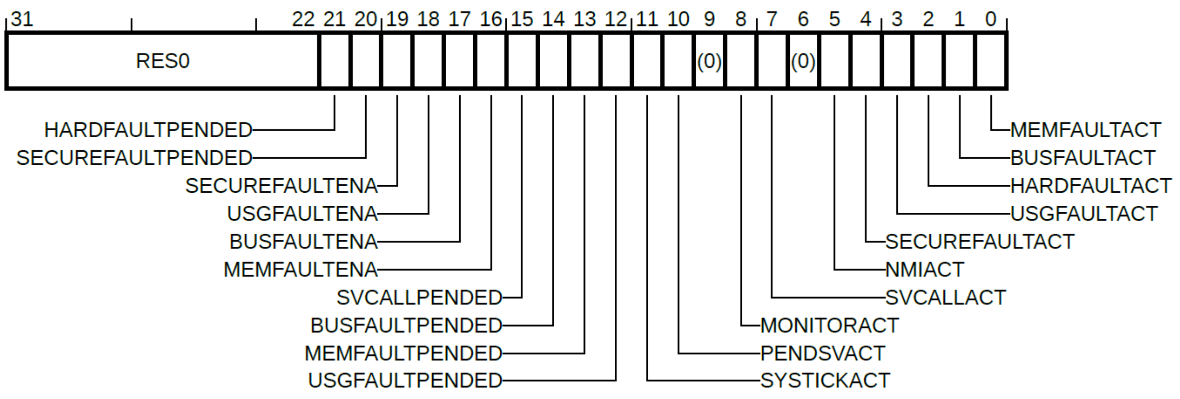
Bits	Name	Description
13	BFHFNMINS	BusFault, HardFault, and NMI enable bit. See Knowledge Article - Software use-case of AIRCR.BFHFNMINS bit for more information.
10:8	PRIGROUP	Priority Grouping. For more information, see Priority grouping
3	SYSRESETREQS	System reset request, Secure only. This bit affects only software-generated reset. It does not affect the debugger's access to the SYSRESETREQ feature.
2	SYSRESETREQ	System Reset Request. When software writes 1 to this bit, the processor requests the system-on-chip to generate a reset.
1	VECTACTIVE	Clears all active state information for exceptions. This is typically used when debugging to allow the system to recover from a system error.

For more information about the AIRCR register bit fields, see Section D1 of the Armv8-M Architecture Reference Manual.

2.10.3 System Handler Control and State Register (SCB->SHCSR)

Use the System Handler Control and State Register (SHCSR) to enable the configurable fault exceptions BusFault, MemManage Fault, UsageFault, and SecureFault by writing to the enable bits in the SCB->SHCSR register. From this register, you can also read the pending and active status for most of the system exceptions including DebugMonitor and NMI exceptions.

Figure 2-13: Bit fields of SHCSR



For more information about the SHCSR register bit fields, see Section D1 of the Armv8-M Architecture Reference Manual

2.11 Special registers for exception masking

There are three special registers that are used to boost the current execution priority, and prevent the current context from being pre-empted. These registers are:

1. PRIMASK. Disables exceptions. For example to enable critical regions in the code to be executed without being interrupted.
2. FAULTMASK. Used by fault exception handlers to suppress the triggering of further faults during fault handling.
3. BASEPRI. Disables exceptions. For example, in some OS operations, it is desirable to block some exceptions for a brief period of time while at the same time, still allowing certain high-priority interrupts to be serviced.

2.11.1 PRIMASK

For many applications, it may be necessary to temporarily disable all the peripheral interrupts to carry out timing-critical task. PRIMASK register can be used under these circumstances. PRIMASK register is only accessible from privileged state. Setting this register boosts the execution priority to 0, and therefore blocks all exceptions with a configurable priority. Hence writing to this register will mask all exceptions except NMI and HardFault.

For C programming, the functions provided in the CMSIS-CORE to set and clear PRIMASK are as follows:

```
void __enable_irq (void);                // Clears PRIMASK
void __disable_irq (void);               // Sets PRIMASK
void __set_PRIMASK(uint32_t priMask);    // Sets PRIMASK to value
uint32_t __get_PRIMASK (void);           // Read PRIMASK value
```

If you are using assembly language programming, then the value of PRIMASK can be changed using the `MRS`/`MSR` or `CPS` instructions as follows:

```
CPSIE I      ; Clears PRIMASK (Enable interrupts)
CPSID I      ; Sets PRIMASK  (Disable interrupts)
ISB          ; Instruction Synchronization Barrier

MOVS R0,#0
MSR PRIMASK,R0   ; Write 0 to PRIMASK to enable exceptions of configurable priority
ISB              ; Instruction Synchronization Barrier

MOVS R0,#1
MSR PRIMASK,R0   ; Write 1 to PRIMASK to disable exceptions of configurable priority
ISB              ; Instruction Synchronization Barrier
```

Because PRIMASK blocks all configurable priority faults, any fault escalates to a HardFault.

2.11.2 FAULTMASK

The behavior of FAULTMASK is very similar to PRIMASK, except that it boosts the priority to -1 and therefore it also blocks HardFault. Some faults can be set to be ignored when the processor is executing at a negative execution priority. FAULTMASK can therefore be used as a way to boost the priority to the point where these faults are ignored. FAULTMASK is often used by the configurable fault handlers MemManage, BusFault, and UsageFault to raise the processor's current priority level. This effectively cause the configurable fault handlers to do the following:

- Bypass the Memory Protection Unit (MPU) settings by setting MPU_CTRL.HFNMIENA bit.
- Ignore the data BusFault for device by setting the Configuration Control Register CCR.BFHFNMIGN bit.

The FAULTMASK register is only accessible in privileged state. When programming with CMSIS-CORE framework, you can use following functions to set and clear a FAULTMASK:

```
void __enable_fault_irq(void);            // Clears FAULTMASK
void __disable_fault_irq(void);           // Set FAULTMASK to disable all exceptions
 except NMI
void __set_FAULTMASK(uint32_t faultMask); // Sets FAULTMASK
uint32_t __get_FAULTMASK(void);           // Reads FAULTMASK
```

If you are using assembly language programming, then the value of PRIMASK can be changed using the `MRS`/`MSR` or `CPS` instructions as shown below:

```
CPSIE F      ; Clears FAULTMASK
CPSID F      ; Sets FAULTMASK
ISB          ; Instruction Synchronization Barrier

MOVS R0,#0
MSR FAULTMASK,R0    ; Write 0 to FAULTMASK to enable exceptions
ISB                 ; Instruction Synchronization Barrier

MOVS R0,#1
MSR FAULTMASK,R0    ; Write 1 to FAULTMASK to disable exceptions
ISB                 ; Instruction Synchronization Barrier
```

FAULTMASK is automatically cleared when returning from an exception handler with a configurable priority.

2.11.3 BASEPRI

In some instances, you might want to disable exceptions with a priority lower than a specified level. In this case, you can use the BASEPRI register. To do this, write the required masking priority level to the BASEPRI register.

For example, if you want to mask all exceptions with a priority level equal or lower than `0x60`, write the following value to BASEPRI:

```
__set_BASEPRI (0x60); // Disables interrupt with priority
                      // 0x60-0xFF using the CMSIS-CORE function
```

```
__isb(void);          // Instruction Synchronization Barrier
```

You can read the value of BASEPRI using the following function:

```
x = __get_BASEPRI(void); // Reads value of BASEPRI
```

If you are using assembly language programming, then the value of BASEPRI can be changed using `MRS/MSR` as follows:

```
MOVS R0,#0x0
MSR BASEPRI,R0      ; Turns off BASEPRI masking
ISB                 ; Instruction Synchronization Barrier
```

The BASEPRI register can be accessed using the BASEPRI_MAX register. The BASEPRI_MAX alias register provides a way of conditionally setting the BASEPRI register if the value being written is a higher priority, that is a lower numerical value, than the current value. Using BASEPRI_MAX can avoid the need to compare the existing value to the required value before setting the register. From the processor hardware perspective, both BASEPRI and BASEPRI_MAX are the same register, but in the assembler code they use different register names. When you use BASEPRI_MAX as a register, the processor hardware automatically compares the current value and new value that is going to be written. It only allows the update if the value is changed to a higher priority level, that is a lower priority value. It cannot be changed to lower priority levels.

For example, consider the following instruction sequence in assembly:

```
MOV R0,#0x60
MSR BASEPRI_MAX,R0    ; Disable interrupts with priority 0x60-0xFF
ISB                   ; Instruction Synchronization Barrier

MOV R0,#0xF0
MSR BASEPRI_MAX,R0    ; This write is ignored because 0xFF is a lower priority level
 than 0x60
ISB                   ; Instruction Synchronization Barrier


MOV R0,#0x40
MSR BASEPRI_MAX,R0    ; This write is allowed to change the masking level to 0x40
ISB                   ; Instruction Synchronization Barrier
```

The following C example shows how to boost the priority level using the BASEPRI register for critical code sections:

```
oldBasePri = __get_BASEPRI();
__set_BASEPRI_MAX(<new requested base pri value>);  // This only boosts BASEPRI.
                                                    // It can never cause the level
 of
                                                    // priority boosting to be
 reduced

<critical code that should not be pre-empted>

__set_BASEPRI(oldBasePri);
```

The BASEPRI and BASEPRI_MAX registers cannot be accessed by unprivileged software. As with other priority level registers, the format of the BASEPRI register is affected by the number of implemented priority register widths. For example, if 3 bits are implemented for priority level registers, then BASEPRI can only be programmed as `0x0`, `0x20`, `0x40` and so on.

See priority-boost-types for another example of how to use BASEPRI to boost priorities.

2.12 Exception handling optimizations

To reduce latency when servicing interrupts, Cortex-M processors implement various optimization techniques. This section discusses some of these techniques are discussed in this section.

2.12.1 Exceptions during multi-cycle instructions

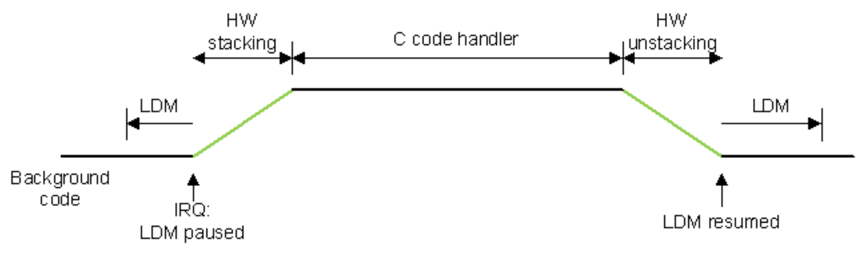
The T32 instruction set includes instructions that can transfer multiple registers to or from memory in a single instruction:

- Load multiple registers: `LDM`, `POP`, and `VLDM`
- Store Multiple registers: `STM`, `PUSH`, `VSTM`

These instructions belong to a specific class of interrupt-continuable instructions.

These instructions can take many cycles to perform all the memory operations. For this reason, it is not desirable to abandon and restart a multiple register transfer. However, waiting for the instruction to complete is also not desirable, as this would add significantly to interrupt latency. So interrupt-continuable instructions record their progress in the ICI field in the EPSR, which is part of the RETPSR that is preserved in the exception stack frame.

Figure 2-14: Interrupt-continuable Load Multiple



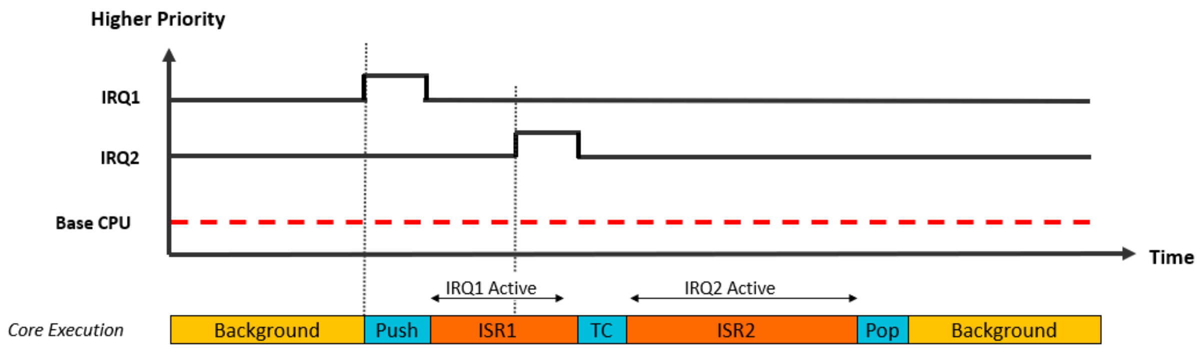
If an exception arrives during execution, multiple register transfers can usually be paused in the middle, and the operation resumed after the exception at the point where it was suspended by the exception. For interrupt-continue, the ReturnAddress is set to the same instruction, as it has not yet completed, but the instruction is resumed in the middle based on the restored EPSR.ICI bits.

Because interrupt-continuable instructions may restart their execution under certain circumstances, using these instructions for Device memory is not recommended.

2.12.2 Tail chaining

When an exception occurs while the processor is handling another exception of the same or of a higher priority, the new exception is pended. When the processor finishes executing the current exception handler, it then proceeds with the pended exception request. Instead of restoring the register's data that was saved on to the stack frame, and then pushing it back again, the processor skips some of stacking and unstacking steps and enters the exception handler of the pended exception as soon as possible. With this arrangement, the timing gap between the two exceptions is considerably reduced.

Figure 2-15: Tail chaining

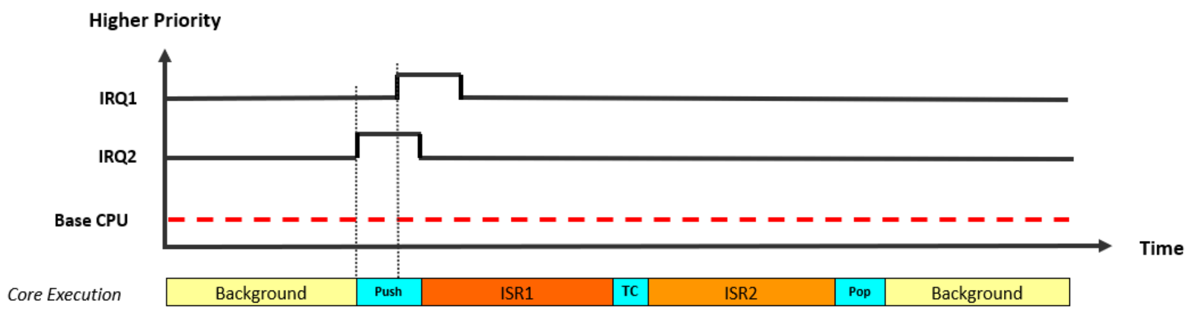


Tail-chaining optimization makes the processor system more energy efficient because the number of memory accesses involved in stacking and unstacking is reduced.

2.12.3 Late arrival

When an exception occurs, the processor accepts the exception request and starts the stacking operation. If, during the stacking operation, another exception of a higher priority occurs, the higher priority late arrival exception is serviced first.

Figure 2-16: Late Arrival



In this example, exception IRQ2 has a lower priority than IRQ1. Exception IRQ2 occurs a few cycles before exception IRQ1. The processor services IRQ1 as soon as the stacking for IRQ2 completes. Therefore the processor can choose to service a late-arriving higher-priority exception before it fetches the vector table.

3. System exceptions

This chapter describes the different types of system exceptions handled by the Cortex-M processor.

3.1 Fault exceptions and their causes

The Armv6-M and Armv8-M Baseline architecture is designed for devices with a small silicon footprint. All fault events occurring in Armv6-M and Armv8-M Baseline processors result in a HardFault exception. HardFault is unrecoverable on these processors because there are no fault status registers to allow the software to determine the cause of the fault exception. The only way to handle the error is to stop the system and perform a self-reset. However, software developers can still analyze errors during software development using debug features including the Micro Trace Buffer (MTB) and Embedded Trace Macrocell (ETM). These features provide recent execution history, enabling investigation of issues. Silicon chip designers can also create their own fault status registers and fault address registers to capture information about bus errors.

The Armv8-M Mainline architecture contains fault status registers. Depending on the processor implementation and which optional features are included, Armv8-M Mainline processors might support several different types of fault exceptions:

Fault Type	Description
HardFault	The default fault exception. Occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism.
BusFault	Occurs because of a memory-related fault for an instruction or data memory transaction. This fault might result from an error that is detected on a bus in the memory system. This error is not available in Armv8-M Baseline or Armv6-M processors.
UsageFault	Caused by an execution error, for example an undefined instruction. This fault exception is not available in Armv8-M Baseline or Armv6-M processors.
MemManage	Caused by either the violation of access permissions set by the Memory Protection Unit (MPU), or attempting to execute code from XN address regions. This fault exception is not available in Armv8-M Baseline or Armv6-M processors.
SecureFault	Caused by Security violations. This fault is available in the Armv8-M Mainline architecture when the Security extension is implemented. In the Armv8-M Baseline architecture with Security extensions, security violations are handled by HardFault.

The BusFault, UsageFault, MemManage, and SecureFault exceptions are configurable fault exceptions. They are disabled by default and can be enabled by software. For more information, see the following:

- See Bus faults for more information about BusFault exceptions.
- See Usage faults for more information about UsageFault exceptions.
- See Memory management faults for more information MemManage exceptions.
- See the Armv8-M Security Extension User Guide for more information about SecureFault exceptions.

Configurable Fault exceptions have programmable exception priority levels similar to interrupts and other system exceptions.

HardFault exceptions can have the following priority levels:

- If the HardFault exception is triggered by a Security violation in Non-secure NMI, the HardFault exception has a priority level of -3, higher than NMI.
- Otherwise, the HardFault exception has a priority level of -1, higher than all other exceptions apart from the NMI.

If a fault event is triggered and the corresponding configurable fault exception is disabled, or if the priority level of the configurable fault exception is not high enough to trigger a pre-emption, the HardFault exception is triggered instead. This is called fault escalation.

3.1.1 Fault status and fault address registers

The Armv7-M and Armv8-M Mainline architectures provide fault status and fault address registers that allow fault handlers to identify the cause of fault exceptions.

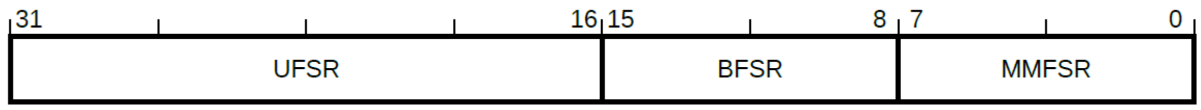
- Fault status registers indicate the cause of a fault.
- Fault address registers indicate the address of the access that triggers the fault. For synchronous BusFaults and MemManage faults, the fault address register indicates the address that is accessed by the operation that caused the fault.

The following table summarizes the available fault status and fault address registers. These registers are accessible only in privileged state.

Address	Register	CMSIS-Core symbol	Function
`0xE000ED28`	Configurable Fault Status Register	SCB->CFSR	Status information for configurable fault exceptions.
`0xE000ED2C`	HardFault Status Register	SCB->HFSR	Status for HardFault exceptions.
`0xE000ED30`	Debug Fault Status Register	SCB->DFSR	Status for Debug events.
`0xE000ED34`	MemManage Fault Address Register	SCB->MMFAR	If available, shows the address that caused the MemManage exception.
`0xE000ED38`	BusFault Address Register	SCB->BFAR	If available, shows the address that caused the BusFault exception.
`0xE000EDE4`	Secure Fault Status Register	SCB->SFSR	If the Security extension is implemented, shows the status for SecureFault exceptions
`0xE000EDE8`	Secure Fault Address Register	SCB->SFAR	If available, shows the address that triggered the SecureFault exception.

The Configurable Fault Status Register (CFSR) is further divided into three parts, as shown in the following figure:

Figure 3-1: Partitioning of Configurable Fault Status Register - CFSR



The CFSR register can be accessed as a whole using a 32-bit data transfer, or each part within the CFSR can be accessed by byte and half word transfers. However, when programming with CMSIS-Core, the only software symbol is 32 bits. There are no CMSIS-Core symbols for the individual MMFSR, BFSR and UFSR parts.

When Halting debug mode is enabled, the vector catch mechanism can be used to generate a debug event and enter Debug state on entry to a specific fault exception. This vector catch mechanism is configured using the enable bits in the Debug Exception and Monitor Control Register (DEMCR).

More information about each fault is provided in subsequent sections.

3.1.2 Memory management faults

MemManage faults can be caused by violation of the access rules which are defined by the MPU configuration. The following are example scenarios that cause MemManage faults:

- Unprivileged tasks trying to access a memory region which is privileged access only.
- Access to a memory location which is not defined by any defined MPU region except PPB space.
- Writing to a memory location which is marked as Read-Only in the MPU.
- Program execution in a memory region which is marked as eXecute Never (XN).

These accesses could be:

- Data accesses during program execution
- Program fetches
- Stack operations during exception sequences

For instruction fetches that trigger a MemManage fault, the fault triggers only when the failed program location enters the execution stage.

For a MemManage fault triggered by stack operations during an exception handling sequence, the following types of errors can occur:

- If the MemManage fault occurred during stack pushing in the exception entry sequence, then it is a stacking error (MSTKERR).
- If the MemManage fault occurred during stack popping in the exception exit sequence, then it is an unstacking error (MUNSTKERR).

- If the MemManage fault occurred during lazy state preservation, then it is a lazy state error (MLSPERR).

See Faults triggered during the Exception handling process for more information about these errors.

Note

The MPU does not control accesses when fetching an entry from the vector table. When the MPU is enabled, accesses to the vector table are always permitted.

A MemManage fault can also be triggered when trying to execute program code in eXecute Never (XN) regions such as Peripheral region, Device region or System region as defined in Section B8.1 of Armv8-M Architecture Reference Manual. Note that this MemManage fault can occur even when without optional MPU.

To understand the details on how to configure an MPU and how MemManage fault works, refer Armv8-M Memory Model and Memory Protection User Guide.

3.1.2.1 MemManage Fault Status Register (MMFSR)

The following table shows the programmer's model for the MemManage Fault Status Register:

CFSR bit	Name	Type	Description	Vector Catch enable bit
7	MMARVALID	Read, write 1 to clear	Indicates that the contents of MMFAR are valid.	-
5	MLSPERR	Read, write 1 to clear	Floating-point lazy-stacking error. Available only when the optional floating-point unit is implemented in a system. A MemManage fault occurring on a lazy state preservation.	DEMCR.VC_INTERR
4	MSTKERR	Read, write 1 to clear	Stacking error. A MemManage fault occurring on an exception entry sequence.	DEMCR.VC_INTERR
3	MUNSTKERR	Read, write 1 to clear	Unstacking error. A MemManage fault occurring on an exception exit sequence.	DEMCR.VC_INTERR
1	DACCVIOL	Read, write 1 to clear	Data access violation during normal code execution.	DEMCR.VC_MMERR
0	IACCVIOL	Read, write 1 to clear	Instruction fetch access violation.	DEMCR.VC_MMERR

Each fault indication status bit, excluding MMARVALID, is set when the fault occurs. The status bit stays high until a value of 1 is written to the register.

The MMARVALID bit in MMFSR is not a fault status indicator. Rather it indicates that it is possible to determine the accessed memory location that caused the fault using the MemManage Fault Address Register, SCB->MMFAR.

Note

In some Cortex-M processors, the fault address registers BFAR, MMFAR, and SFAR could be implemented as a single shared register. In this situation, raising one exception might invalidate the fault address register for a previous exception. For example, a BusFault exception might invalidate the MMFAR register for a previous MemManage exception. In this case, the processor clears the MMARVALID bit.

3.1.2.2 Programming MemManage using the CMSIS framework

In Armv8-M Mainline processors, MemManage faults can optionally be enabled. However, before enabling the MemManage fault handler, the exception priority levels should be configured. When using the CMSIS-Core framework, the following functions program the priority level of the fault:

```
NVIC_SetPriority(MemoryManagement_IRQn,<priority>);
```

The enable control bits for the fault handlers are in the System Handler Control and State register (SCB->SHCSR). To enable a fault exception, setting the corresponding enable bit to 1. For example, to enable MemManage exceptions do the following:

```
SCB->SHCSR |= SCB_SHCSR_MEMFAULTENA_Msk; // Set SHCSR[16] = 1
```

The MemManage fault handler is declared in the vector table startup code as follows:

```
void MemMange_Handler(void)
```

This handler is defined with a `weak` C attribute, so it is overridden if you define a new fault handler.

3.1.3 Bus faults

Bus faults can be triggered by error responses received from the processor bus interface during memory accesses. Bus faults can occur in the following situations:

- Instruction fetch
- Data read or write

Bus faults can occur during stacking or unstacking of an exception handling sequence:

- If a bus error occurs during an exception entry sequence, it is called a stacking error, STKERR.
- If a bus error occurs during an exception exit sequence, it is called an unstacking error, UNSTKERR.

- If a bus error occurs during lazy stacking, it is called as lazy stack error, LSPERR. Lazy stacking is only available when the Floating-point extension is implemented and enabled.

See Faults triggered during the Exception handling process for more information about how faults are handled on exception entry, exception exit, and during lazy state preservation sequences.

If a bus error occurs while fetching an entry from the vector table, a HardFault exception is raised even when the BusFault exception is enabled.

There are several reasons why a bus error can occur. For example:

- When the processor attempts to access an invalid memory location. In this case, the bus responder might return an error response resulting in BusFault.
- When the device is not ready to accept a transfer. For example, when the device tries to access DRAM without initializing the DRAM controller.
- When the bus responder receiving the transfer request returns an error response.
- When unprivileged software accesses a privileged-access only register on a Private Peripheral Bus (PPB).

Bus faults are subdivided into two classes: synchronous and asynchronous bus faults.

- Synchronous bus fault

  This bus fault is also referred as a precise bus fault. A synchronous bus fault refers to a fault exception that can be attributed to the instruction that caused it. Synchronous BusFaults are subject to the escalation process.
- Asynchronous bus fault

  This bus fault is also referred as an imprecise bus fault. An asynchronous bus fault refers to a fault exception that cannot be attributed to the instruction that caused it, where the processor could have executed further instructions before the exception sequence started.

  When an asynchronous bus fault is triggered, the BusFault exception is pended. If another higher priority interrupt event arrived at the same time, the higher priority interrupt handler is executed first, and then the BusFault exception takes place. If the BusFault handler is not enabled, a HardFault exception is pended instead. A HardFault caused by an asynchronous BusFault never escalates into lockup.

3.1.3.1 Bus Fault Status Register (BFSR)

The following table shows the programmer's model for the Bus Fault Status Register:

CFSR bit	Name	Type	Description	Vector Catch enable bit
15	BFARVLID	Read, write 1 to clear	Indicates that BFAR is valid.	-

CFSR bit	Name	Type	Description	Vector Catch enable bit
13	LSPERR	Read, write 1 to clear	Floating-point lazy-stacking error. Available only when the optional floating-point unit is implemented in a system. BusFault occurring on a lazy state preservation.	DEMCR.VC_INTERR
12	STKERR	Read, write 1 to clear	Stacking error. BusFault occurring on an exception entry sequence.	DEMCR.VC_INTERR
11	UNSTKERR	Read, write 1 to clear	Unstacking error. BusFault occurring on an exception exit sequence.	DEMCR.VC_INTERR
10	IMPRECISERR	Read, write 1 to clear	Imprecise data access error. Usually, a bus fault becomes asynchronous, or imprecise, when the processor has write buffers or caches.	DEMCR.VC_BUSERR
9	PRECISERR	Read, write 1 to clear	Precise data error. For example, a synchronous, or precise, bus fault can occur if the response from the bus responder returns an error.	DEMCR.VC_BUSERR
8	IBUSERR	Read, write 1 to clear	Instruction access error.	DEMCR.VC_BUSERR

Each fault indication status bit, excluding BFARVALID, is set when the fault occurs. The status bit stays high until a value of 1 is written to the register.

When the BFSR indicates that a fault is a synchronous bus error by PRECISERR, or an instruction access bus error by IBUSERR, the faulting code address is usually reflected by the stacked program counter in the stack frame.

Note

There is no bit field in BFSR to indicate a failed exclusive access operation.

3.1.3.2 Programming BusFault using the CMSIS framework

In Armv8-M Mainline processors, BusFault faults can optionally be enabled. However, before enabling the BusFault fault handler, the exception priority levels should be configured. When using the CMSIS-Core framework, the following functions program the priority level of the fault:

```
NVIC_SetPriority(BusFault_IRQn,<priority>);
```

The enable control bits for the fault handlers are in the System Handler Control and State register (SCB->SHCSR). To enable a fault exception, setting the corresponding enable bit to 1. For example, to enable BusFault exceptions do the following:

```
SCB->SHCSR |= SCB_SHCSR_BUSFAULTENA_Msk; // Set SHCSR[17] = 1
```

The BusFault fault handler is declared in the vector table startup code as follows:

```
void BusFault_Handler(void)
```

This handler is defined with a `weak` C attribute, so it is overridden if you define a new fault handler.

3.1.4 Usage faults

There are a wide range of reasons for a Usage Fault exception.

The following table identifies several example scenarios that could cause a UsageFault exception, and shows the programmer's model for the Usage Fault Status Register:

CFSR bits	Name	Description	Vector Catch enable bit
25	DIVBYZERO	Indicates a divide by zero has taken place. This bit can only be set if DIV_O_TRP is set.	DEMCR.VC_CHKERR
24	UNALIGNED	An unaligned memory access for a Load/Store multiple instruction.	DEMCR.VC_CHKERR
20	STKOF	Stack Overflow flag. Occurs when there is a stack limit violation.	DEMCR.VC_INTERR
19	NOCP	An attempt to execute a coprocessor instruction, for example floating-point instructions in the FP or MVE extensions, or an Arm Custom Instruction when the coprocessor or Arm Custom Instruction is either not present, disabled, or not accessible. See Floating-point and MVE support for more information.	DEMCR.VC_NOCPERR
18	INVPC	An attempt to carry out an exception return with an invalid value for EXC_RETURN.	DEMCR.VC_STATERR
17	INVSTATE	An attempt to switch to an invalid state. For example, since Cortex-M processors only support Thumb instructions, if software is ported from another Arm processor, which could contain Arm instructions, this would result in a UsageFault. Another example is if an exception returns with Interrupt-Continuable Instruction (ICI) bits in the unstacked xPSR, but the instruction being executed after the exception return is not a multiple load/store instruction.	DEMCR.VC_STATERR
16	UNDEFINSTR	An attempt to execute an undefined instruction.	DEMCR.VC_STATERR

Each fault indication status bit is set when the fault occurs. The status bit stays high until a value of 1 is written to the register.

3.1.4.1 Programming UsageFault using the CMSIS framework

In Armv8-M Mainline processors, UsageFault faults can optionally be enabled. However, before enabling the UsageFault fault handler, the exception priority levels should be configured. When using the CMSIS-Core framework, the following functions program the priority level of the fault:

```
NVIC_SetPriority(UsageFault_IRQn,<priority>);
```

The enable control bits for the fault handlers are in the System Handler Control and State register (SCB->SHCSR). To enable a fault exception, setting the corresponding enable bit to 1. For example, to enable UsageFault exceptions do the following:

```
SCB->SHCSR |= SCB_SHCSR_USGFAULTENA_Msk; // Set SHCSR[18] = 1
```

The UsageFault fault handler is declared in the vector table startup code as follows:

```
void UsageFault_Handler(void)
```

This handler is defined with a `weak` C attribute, so it is overridden if you define a new fault handler.

3.1.5 Secure faults

The SecureFault exception is triggered by violations of security rules. For more information about these security rules, see the Armv8-M Architecture Reference Manual. If the Security extension is not implemented, then SecureFault exceptions are not available.

Here are some examples of security violations that can trigger a SecureFault exception:

- Memory accesses that violate security permissions.
- An illegal transition between security states. For example, a branch from the Non-secure world to the Secure world without going through a valid entry gateway.
- When a security integrity check fails during an exception sequence. For example, an invalid EXC_RETURN value.

Note

At the system level, bus accesses could be filtered by TrustZone security components such as the Memory Protection Controller or the Peripheral Protection Controller. These components trigger a BusFault using bus error responses, not a SecureFault.

See the Armv8-M Security Extension User Guide for more information about SecureFault causes, the SecureFault Status Register (SFSR), and the SecureFault Address Register (SFAR).

3.1.5.1 Programming SecureFault using the CMSIS framework

In Armv8-M Mainline processors, SecureFault faults can optionally be enabled. However, before enabling the SecureFault fault handler, the exception priority levels should be configured. When using the CMSIS-Core framework, the following functions program the priority level of the fault:

```
NVIC_SetPriority(SecureFault_IRQn,<priority>);
```

The enable control bits for the fault handlers are in the System Handler Control and State register (SCB->SHCSR). To enable a fault exception, setting the corresponding enable bit to 1. For example, to enable SecureFault exceptions do the following:

```
SCB->SHCSR |= SCB_SHCSR_SECUREFAULTENA_Msk; // Set SHCSR[19] = 1
```

The SecureFault fault handler is declared in the vector table startup code as follows:

```
void SecureFault_Handler(void)
```

This handler is defined with a `weak` C attribute, so it is overridden if you define a new fault handler.

3.1.6 HardFaults

A HardFault exception can be triggered by the following situations:

- Receiving a bus error response for a vector fetch.
- A Security violation for a vector fetch.
- Executing an SVC instruction when the priority level of the SVC exception is the same or lower than the current priority level.
- Executing a BKPT breakpoint instruction when debug is disabled.
- When the configurable faults including MemManage, BusFault, UsageFault, and SecureFault are not enabled, then they are escalated to HardFault.

3.1.6.1 HardFault Status Register (HFSR)

The following table shows the programmer's model for the HardFault Status Register:

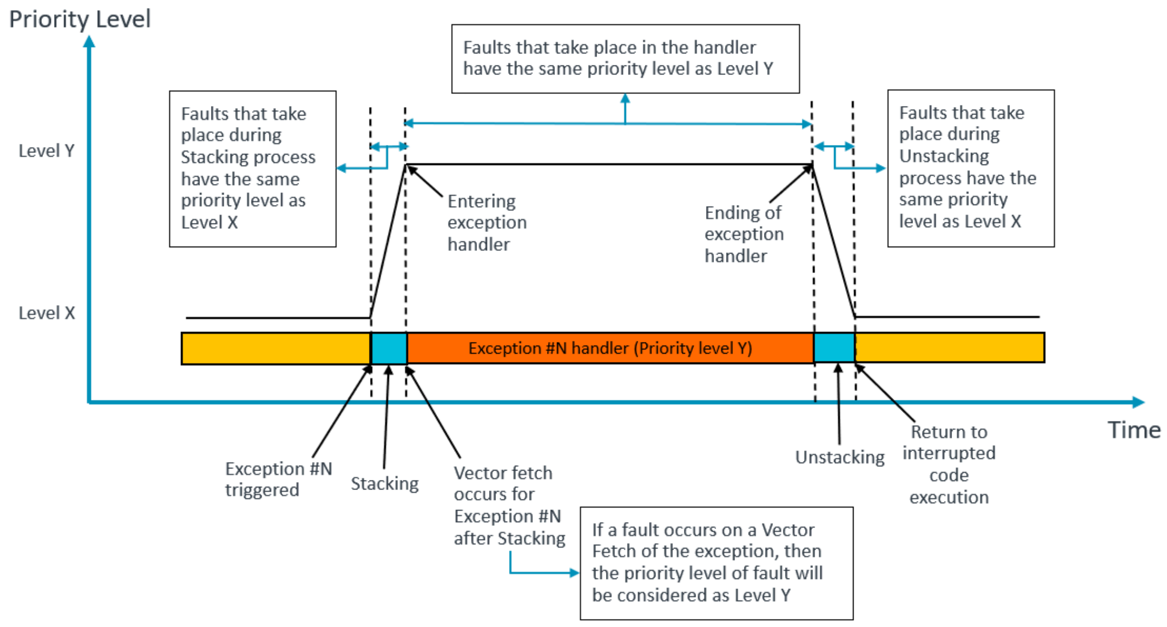
HFSR bits	Name	Description	Vector Catch Enable bit
31	DEBUGEVT	Indicates that a HardFault has been triggered by a debug event.	DEMCR.VC_HARDERR
30	FORCED	Indicates that a fault has been escalated from one of the configurable faults. The fault handler should check the CFSR bit for the fault cause.	DEMCR.VC_HARDERR
1	VECTBL	Indicates that a HardFault has been triggered by a failed vector fetch.	DEMCR.VC_INTERR

Each fault indication status bit is set when the fault occurs. The status bit stays high until a value of 1 is written to the register.

3.1.7 Faults triggered during the Exception handling process

If a stacking or unstacking error occurs during an exception sequence, the priority level for error handling is based on the priority level of the interrupted code. The following diagram shows the process:

Figure 3-2: Priority of stacking and unstacking faults


Priority Level
Level Y
Level X
Time
Faults that take place in the handler have the same priority level as Level Y
Faults that take place during Stacking process have the same priority level as Level X
Faults that take place during Unstacking process have the same priority level as Level X
Entering exception handler
Ending of exception handler
Exception #N handler (Priority level Y)
Exception #N triggered
Stacking
Vector fetch occurs for Exception #N after Stacking
If a fault occurs on a Vector Fetch of the exception, then the priority level of fault will be considered as Level Y
Unstacking
Return to interrupted code execution

In the diagram, level X indicates the level of the interrupted code and level Y indicates the priority level of the exception to be serviced.

The diagram shows that if a stacking or unstacking error occurs during an exception sequence, the following scenarios are possible:

1. If the configurable fault exception is disabled, or has the same or lower priority level than the priority Level X, then it is immediately escalated to HardFault exception. Configurable faults are BusFault, MemManage, SecureFault, and UsageFault.
2. If the fault exception is enabled and has a higher priority level than both Level X and Level Y, the fault exception is executed first and exception #N is pended.
3. If the fault exception is enabled and has a priority level between Level X and Level Y, then the handler for exception #N is executed first. The triggered fault is serviced later.
4. A fault occurs during lazy stacking. If the FPU is implemented and enabled, and if the exception handler #N uses the FPU, then the stacking of FPU registers happens later during the execution of exception handler #N. For more information, see Lazy Floating-point State preservation. In the lazy stacking scenario, if the memory access for lazy stacking triggers a fault event, it is handled as though the fault had occurred during stacking process. For example:

- If the configurable fault exception is disabled, or has the same or a lower priority level than priority level X then it escalates to HardFault.

- If the configurable fault exception is enabled and has a higher priority level than level Y, then the configurable fault exception executes immediately.
- If the configurable fault exception is enabled and has the same or a lower priority level than level Y, then the pending status of the configurable fault exception is set and executes when exception handler #N has finished.

3.1.8 Using FAULTMASK in a configurable fault handler

Setting FAULTMASK disables all interrupts except the NMI exception. The configurable fault handlers BusFault, MemManage, UsageFault, and SecureFault can all utilize the FAULTMASK feature to do the following:

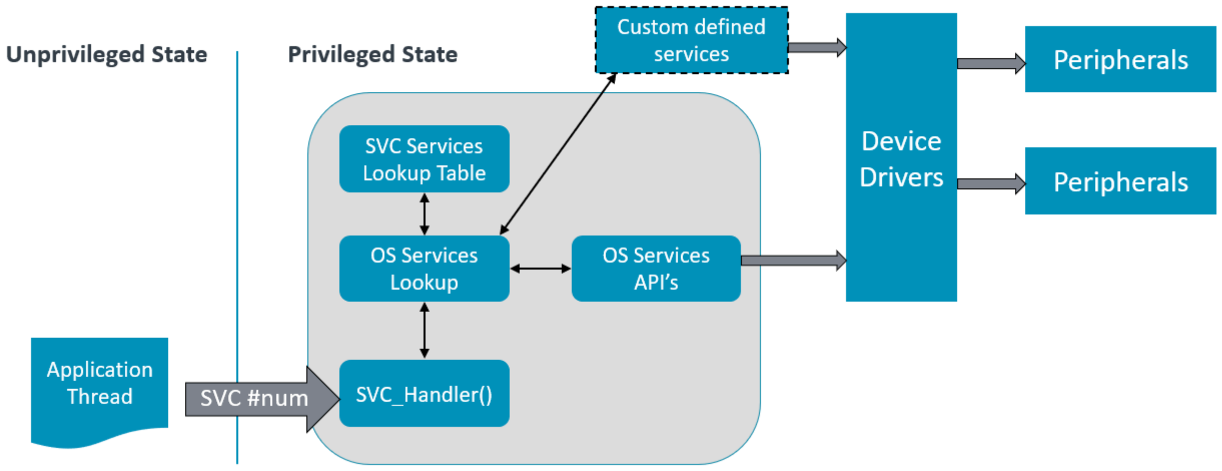
Purpose	Description
Disable the MPU in handlers with -1 priority, for example when HardFault or FAULTMASK is set to 1, using the HFNMIENA bit in the MPU Control register, MPU_CTRL.	The MPU_CTRL.HFNMIENA bit defines the behavior of the MPU during the execution of NMI and HardFault handlers. By default, MPU_CTRL.HFNMIENA is set to 0, meaning that the MPU is disabled in handlers at priority -1 or -2. This allows both HardFault and NMI handlers to execute critical code even when the MPU has been incorrectly configured.
Suppress stack limit checking using the Stack Overflow HardFault NMI Ignore (STKOFHFNMIGN) bit in the Configuration Control Register (CCR)	The CCR.STKOFHFNMIGN bit allows handlers with -1 priority to bypass stack limit checks. When using the stack limit check feature, the CCR.STKOFHFNMIGN bit is useful when you want to reserve some memory space at the end of the main stack for the HardFault and NMI handlers.
Suppress bus faults using the HardFault NMI Ignore (BFHFNMIGN) bit in the CCR register	The CCR.BFHFNMIGN bit prevents data access bus faults when the processor is executing at priority -1 or -2. An example use of the bit is in autoconfiguration of a bridge or other device, where probing a disabled or non-existent element might cause a bus fault. Before using this bit, ensure that the code and data spaces of the handler that executes at priority -1 or -2 are valid for correct operation. Note that from Armv8.1-M onwards, the CCR.BFHFNMIGN bit is deprecated.

3.2 Supervisor Call - SVC

Operating System support is provided by the Supervisor Call instruction, `svc`. Because this instruction can be executed in unprivileged Thread mode, it provides a method by which unprivileged applications can request privileged operations. The architecture also provides a closely related exception called the Pended Supervisor call, PendSV, that can be used to avoid the need to boost priority around critical sections of OS code. See Pended SVC - PendSV for more details.

The `svc` instruction includes an 8-bit field that can be set to an arbitrary value to distinguish between different privileged services that might be provided by the OS. The architecture does not provide a mechanism to make this immediate value directly visible in the handler, so to use this value, software must perform a slightly indirect sequence to obtain it. The sequence involves examining the EXC_RETURN token to identify which stack contains the exception stack frame, reading the ReturnAddress to identify from where the SVCall was invoked, and reading the byte before the address indicated by ReturnAddress to obtain the embedded immediate value.

Figure 3-3: SVCall gateway for OS services



An alternative method for identifying different functions in an SVC handler is for the calling code to pass parameters to the stacked versions of registers used for SVCall.

Note

Accessing the register contents directly from the SVC gives erroneous results and might fail because of late-arriving interrupts. Instead, the argument values must be read from the stacked versions of the registers used for the SVC exception sequence.

SVCall is a unique exception in having this ability. Because the SVC occurs at a known point in code, it can be preceded by assigning values to registers so that the handler code is able to see these parameter values. This aligns with the AAPCS compiler rules specifying that registers r0 to r3 are used for parameter passing to function calls, so in high level languages, the identifier for the required function can simply be an input parameter. This also has the potential advantage of providing more than the 256 unique values that are available in the 8-bit opcode field.

A typical SVC handler does the following:

- Extract the SVC service number from the program memory. To do this, extract the stacked PC in the stack frame and then use this value to read the SVC number.
- Depending on the SVC service being accessed, extract arguments or parameters from the stack frame.
- Depending on the SVC service being accessed, return the results in the stack frame.

To allow the SVC service to directly manipulate the stack memory, the SVC handler needs an assembly wrapper. This assembly wrapper collects the following information, which is then passed to SVC handler as function arguments:

- The value of EXC_RETURN
- The value of the stack pointer used by the background code.

The following code provides a simple example of an assembly wrapper for an SVC handler in C:

```
void SVC_Handler(void)
{
  __asm(
    ".global SVC_Handler_Main\n"
    "TST lr, #4\n"
    "ITE EQ\n"
    "MRSEQ r0, MSP\n"
    "MRSNE r0, PSP\n"
    "B SVC_Handler_Main\n"
  ) ;
}
```

Both functions end with `SVC_Handler_Main()`, which is a normal C function. This function looks at the instruction before the stacked return address to determine the SVC number. The `SVC 0` instruction in this example disables privileged mode.

```
void SVC_Handler_Main( unsigned int *svc_args )
{
  unsigned int svc_number;

  /*
  * Stack contains:
  * r0, r1, r2, r3, r12, r14, the return address and xPSR
  * First argument (r0) is svc_args[0]
  */
  svc_number = ( ( char * )svc_args[ 6 ] )[ -2 ] ;
  switch( svc_number )
  {
    case 0:  /* EnablePrivilegedMode */
      __set_CONTROL( __get_CONTROL( ) & ~CONTROL_nPRIV_Msk ) ;
      break;
    default:    /* unknown SVC */
      break;
  }
}
```

For more details about the actual code sequence, see the svc-number-as-parameter use case example.

Because of the nature of exception-handling mechanisms, when using `SVC` software designers must consider the following:

- The `SVC` instruction should not be used in an exception or interrupt service routine that has the same or higher group priority than the SVCall exception, or when an interrupt masking register is set that blocks the SVCall exception. If the SVCall exception cannot be executed, a HardFault exception is triggered.
- When passing parameters to an SVC service using registers r0-r3, the SVC service needs to extract these parameters from the exception stack frame rather than take the current values of these registers in the register bank. This is because if another interrupt service executes just before the SVC handler and is tail-chained into the SVC service, the values in r0-r3 and r12 might be changed by the previous interrupt service routine.
- If an SVC service needs to return a value back to the calling task, the return value should be written to the exception stack frame so that it can be read back into r0-r3 on exception exit.

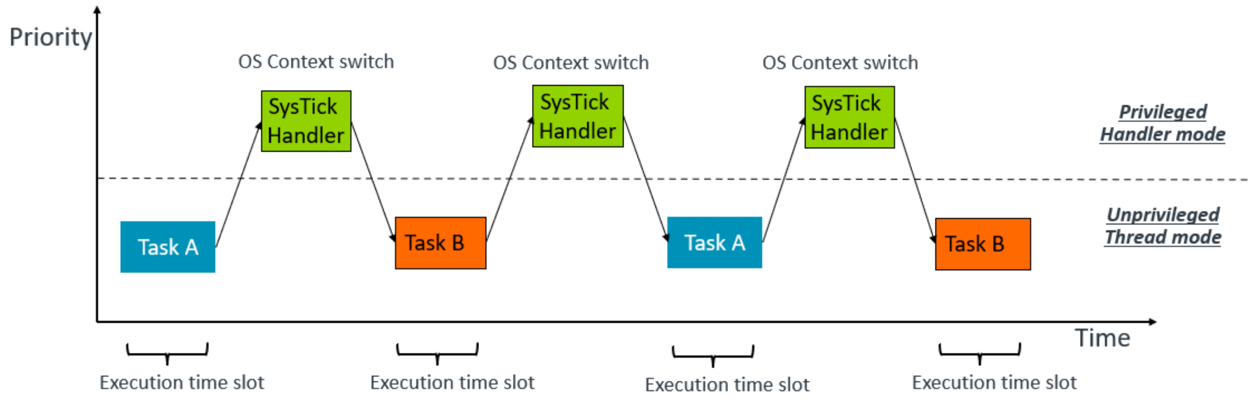
3.3 Pended SVC - PendSV

The PendSV feature allows software to trigger an exception. Like IRQs, PendSV is asynchronous. An embedded OS or RTOS can use the PendSV exception to defer processing tasks instead of using an IRQ. Unlike IRQs, where the exception number assignments are device-specific, the same PendSV control code can be used by all Arm Cortex-M processors. This is because the PendSV exception is part of the architecture and present on all Cortex-M processors. PendSV therefore allows an embedded OS or RTOS to run out-of-the-box on all Cortex-M based systems without customization. The PendSV exception is not invoked by a specific instruction, but rather by privileged software setting ICSR.PENDSVSET to 1'b1.

In an RTOS environment, PendSV is usually configured to have the lowest interrupt priority level. The OS kernel code, which executes in Handler mode at high priority level is therefore able to schedule some OS operations using PendSV, to be carried out at a later time. By using PendSV, those deferred OS operations can be carried out at the lowest exception priority level when no other exception handler is running. One of these deferred OS operations is an OS context switch, which forms an essential part of a multitasking system.

The following diagram shows the basic concept of context switching:

Figure 3-4: Simple context switching operation



In a simple OS design, the execution time is divided into number of time slots. For a system with two OS tasks, an OS might execute those tasks alternately. In this example, a SysTick timer is used to generate a periodic exception to trigger context switching. For more information, refer to the example rtos_context_switch project source code available on GitHub.

In Cortex-M processors, OS designers can separate the context switching operation from the SysTick handler by placing the operation into a separate PendSV exception handler. Using separate exceptions in this way has the following benefits:

- The SysTick handler, which handles the task scheduling evaluation, can still run at a high priority level and if there is no other interrupt service running, context switching can be performed.

- The PendSV exception, which is triggered by the OS scheduling code, can run at the lowest priority level and carry out the deferred context switching when needed. This is needed when the OS code in the SysTick handler needs to carry out a context switch but has detected that the processor is servicing another interrupt.

Note

Why is the PendSV exception preferred for context switching operations?

Using PendSV at the lowest interrupt priority avoids situations where context switching could occur in the middle of executing an ISR. Because PendSV is set to the lowest priority, when it is executing there can never be other ISR running in the background and therefore the context switch will only suspend the current background thread. Although you could use SysTick for simple context switching operations, if there are interrupt routines that have higher priority than SysTick, then saving callee-saved registers becomes difficult. Therefore it is preferable to use PendSV for context switching operations, and SysTick for other OS operations within the system.

The following table describes the key differences between SVCall and PendSV exceptions:

Characteristic	SVCall	PendSV
How is it used in an OS environment?	Allows unprivileged threads access to privileged OS services.	Handles context switching.
Triggering mechanism	Execution of `SVC` instruction.	Sets its pending status by writing to SCB->ICSR.
Priority level	Programmable.	Programmable. Typically, in an OS environment, it is set to the lowest priority level.
CMSIS-Core handler name	`void SVC_Handler(void)`	`void PendSV_Handler(void)`
Exception nature	Synchronous. After execution of the `SVC` instruction, subsequent instructions in the current context are not allowed to execute until the SVC handler has been taken.	Asynchronous. After setting the PendSV status bit, the processor is able to execute additional instructions in the current context before the PendSV handler executes. Note that this can only be guaranteed if the PendSV handler is lower priority than the code that set the pending status bit.

3.4 SysTick

The architecture defines a timer function, SysTick, that provides a standardized timer that can be used for OS scheduling or other purposes. SysTick has a special calibration feature that allows the system designer to provide software-visible information about the clocking on the chip so that platform-independent software has a standard way of calculating accurate real-time intervals using the SysTick timer on any chip that provides this data.

The SysTick timer is optional in Armv8-M Baseline and mandatory in Armv8-M Mainline.

In Armv8-M Baseline with the security extension configured, the number of instances of SysTick can be as follows:

- 0, not configured.
- 1, a single timer, assigned to either security state at runtime by software programming the STTNS bit in the Interrupt Control and Stare Register, ICSR.STTNS.
- 2, a separate instance in each security state.

In Armv7-M and Armv8-M Mainline, SysTick is a mandatory feature. If the security extension is implemented there is a separate instance of SysTick for each security state.

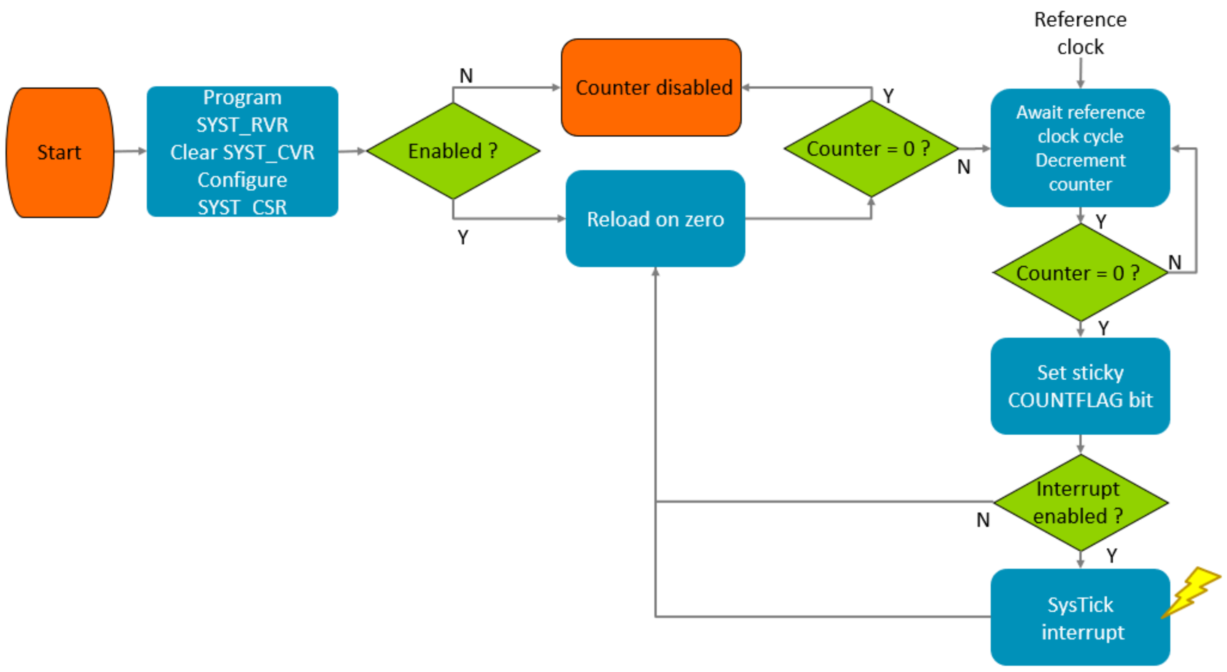
3.4.1 Basic SysTick Timer Operation

The SysTick timer is a simple 24-bit timer and contains four registers. The following table shows these four registers:

Address	CMSIS-Core symbol	Register description
`0xE000E010`	SysTick->CTRL	SysTick Control and Status Register
`0xE000E014`	SysTick->LOAD	SysTick Reload Value Register
`0xE000E018`	SysTick->VAL	SysTick Current Value Register
`0xE000E01C`	SysTick->CALIB	SysTick Calibration Register

The timer operates as a down counter and triggers the SysTick exception when it reaches 0, as shown in the following diagram. After reaching zero, at the next transition, the timer automatically reloads using the value in the reload value register. The timer can run at the processor's clock frequency but can also be set up to decrement using a reference clock, if available.

Figure 3-5: SysTick operation details



When SysTick is enabled, that is SysTick->CTRL.ENABLE is set to 1, the down counter register SysTick->VAL decrements as follows:

- At the processor's clock speed if SysTick->CTRL.CLKSOURCE is 1
- At the rising edge of a reference clock if SysTick->CTRL.CLKSOURCE is 0.

In some devices, if there is no reference clock available, then the NOREF bit in SysTick->CALIB should be set to 1.

When the down counter value in SysTick->VAL counts down to 0, the SysTick->CTRL.COUNTFLAG is automatically set to 1 by the processor. If SysTick->CTRL.TICKINT is set, then the SysTick exception is pended. The reload value in SysTick->LOAD is then loaded into the SysTick's counter value in SysTick->VAL. The COUNTFLAG is not cleared until it is explicitly cleared either by a register read or when SysTick->VAL is cleared.

Note

When the processor enters sleep mode, the processor's clock is stopped and therefore the SysTick timer is also stopped.

The SysTick Calibration Register, SysTick->CALIB, allows the on-chip hardware to provide calibration information to software as follows:

- The TENMS field of the SysTick->CALIB register provides the reload value required to achieve a SysTick interval of 10ms.
- The NOREF bit in the SysTick->CALIB register indicates whether a reference signal is provided, in which case software can choose whether to count reference cycles or CPU clock cycles.
- The SKEW bit in SysTick->CALIB acts as an indicator to software to identify whether the TENMS calibration value is a precise integer, or a fractional number rounded to the nearest integer.

In CMSIS-Core, the use of SysTick Calibration Register is normally not needed because CMIS-Core provides a software variable called `SystemCoreClock`. This variable is configured in the system initialization function `SystemInit()` and is also updated each time the system clock configuration is changed. With the CMSIS-Core framework, the SysTick exception handler is called as `SysTick_Handler(void)`.

3.4.2 Using the SysTick timer

The CMSIS-Core header file provides a function that generates periodic SysTick interrupts using the processor's clock as a clock source:

```
uint32_t SysTick_Config (uint32_t ticks);
```

This function sets the SysTick interrupt interval to `ticks`, enables the counter using the processor clock, and enables the SysTick exception with the lowest exception priority.

For example, if you have a clock frequency of 30 MHz, and you want to trigger SysTick exceptions with a frequency of 1KHz, you can use any one of the following function calls:

```
SysTick_Config(SystemCoreClock/1000);  // SystemCoreClock = 30 x 10^6;

or

SysTick_Config(30000);                 // 30MHz/1000 = 30000
```

The `SysTick_Handler` then triggers at a rate of 1 KHz.

You can also manually create SysTick timer setup code. See the SysTick example project source code for more information.

3.4.3 Using the SysTick timer for timing measurement

The SysTick timer can be used for timing measurements. For example, you can measure the duration of short functions using the following code sequence:

```
unsigned int start_time, stop_time, cycle_count;

SysTick->CTRL = 0;          // Disable SysTick
SysTick->LOAD = 0xFFFFFFFF; // Set the reload value to the maximum
SysTick->VAL  = 0;          // Clear the current value to 0
SysTick->CTRL = 0x5;         // Enable SysTick and use the processor clock
__dsb();
__isb();

while (SysTick->VAL != 0)  // Wait until SysTick is reloaded
start_time = SysTick->VAL; // Obtain the start time

function()                 // Execute the function to be measured

stop_time = SysTick->VAL;  // Obtain the stop time

cycle_count = start_time - stop_time; // Calculate the time taken
```

Because SysTick is a decrementing counter, the value of `start_time` is greater than `stop_time`. If the execution time of the function being measured is very high, that is more than $2^2$4 clock cycles, then the timer would underflow. Therefore there might be a need to include a check the value of `count_flag` at the end of the timing measurement. If `count_flag` is set, the duration being measured will be more than `0xFFFFFF` clock cycles. In that case, the SysTick Handler used to count how many times the SysTick counter has underflowed. The total number of clock cycles measured would then also include the SysTick exception.

3.5 Escalations

All fault exceptions except for HardFault have a configurable exception priority. Software can disable execution of the configurable fault handlers, but not the HardFault handler.

Usually, the exception priority and the values of the exception enable registers determine whether the processor enters the fault handler. The exception priority registers determine whether one fault handler can pre-empt another fault handler.

In some situations, a fault with configurable priority is treated as a HardFault. This situation is called priority escalation, and the fault is described as being escalated to HardFault.

Escalation to HardFault occurs when:

- A fault handler causes the same kind of fault as the one it is servicing. The escalation to HardFault occurs because a fault handler cannot pre-empt itself because it must have the same priority as the current priority level.
- A fault handler causes a fault with the same or lower priority as the fault it is servicing. This is because the handler for the new fault cannot pre-empt the currently executing fault handler.
- An exception handler causes a fault for which the priority is the same as or lower than the currently executing exception.
- A fault occurs and the handler for that fault is not enabled.

3.5.1 Lockup

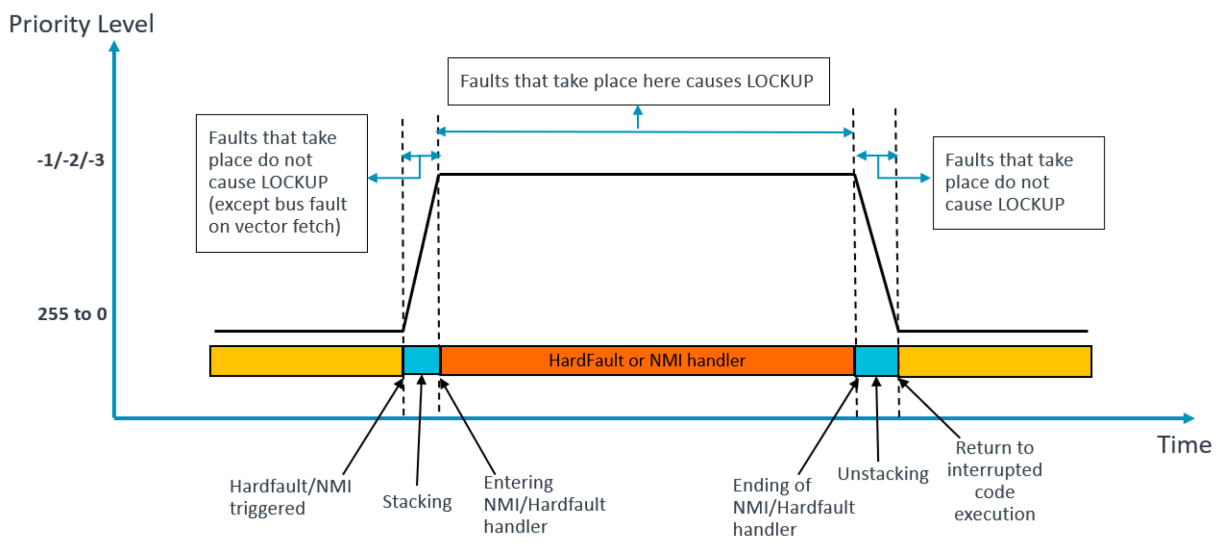
The processor enters into the lockup state if a fault escalation occurs when executing at negative priority. Lockup can occur in the following situations:

- A fault occurs during the execution of the HardFault or the NMI exception handler
- A bus error occurs during the vector fetch for the HardFault or NMI exception
- The SVC instruction is accidentally included in the HardFault or NMI exception
- The vector fetch occurs during the startup sequence

When the processor is in lockup state, it does not execute any instructions. It asserts an output signal called LOCKUP. How this output signal is used inside the system chip depends on the system level design. However, in some cases, it can be used to automatically generate a system reset.

If the lockup is caused by a fault event inside the HardFault handler at priority level -1, it is still possible for the processor to respond to an NMI and execute the NMI handler. But, after the NMI handler finishes, it will return to the lockup state and the priority level will return to -1.

Figure 3-6: Faults that cause lockup


Priority Level
Faults that take place here causes LOCKUP
-1/-2/-3
Faults that take place do not cause LOCKUP (except bus fault on vector fetch)
Faults that take place do not cause LOCKUP
255 to 0
HardFault or NMI handler
Time
Hardfault/NMI triggered
Stacking
Entering NMI/Hardfault handler
Ending of NMI/Hardfault handler
Unstacking
Return to interrupted code execution

The processor remains in lockup state until any one of the following wake-up events are triggered:

- Reset
- An NMI interrupt, if the processor is locked up at -1 priority
- Halt by a debugger

Note the following:

1. If the lockup state occurs from the NMI handler, a subsequent NMI does not cause the processor to leave lockup state.
2. When entering a HardFault or NMI handler, a fault that occurs during stacking or unstacking, for example a bus error or an MPU access violation, does not cause the system to enter into a lockup state. If a bus error is triggered during stacking, the BusFault exception would become pended and only executes after the HardFault handler has finished.

3.5.2 How to avoid lockup

In some applications, it is important to avoid lockup and therefore care needs to be taken when writing HardFault and NMI handlers.

Here are few points to consider when writing HardFault and NMI handlers:

1. Stack pointer checks should be placed at the beginning of the handlers, to ensure that the MSP is within the valid range.
2. It is good practice to partition the exception handling tasks such that HardFault and NMI handlers only carry out essential tasks.
3. Ensure that NMI and HardFault handler code does not call functions that use the `svc` instruction. In some software designs, high level message output functions such as error

reporting could be redirected to OS functions such as semaphore calls. Using those OS functions in the HardFault or NMI handler result in lockup state because the SVC exception is always of lower priority level than the HardFault or NMI handler.

4. Floating-point and MVE support

This section describes the Floating-point (FP) and M-Profile Vector Extension (MVE) support mechanisms. The MVE extension to the Armv8.1-M architecture is also commonly referred to as Helium technology.

4.1 Introduction

In high level languages like C and C++, floating-point numerical values can be represented using the `float` and `double` data types. The floating-point data types allow the processor to handle a much wider data range compared to integers or fixed-point data, as well as very small values. The Institute of Electrical and Electronics Engineers (IEEE) produced standards for the encoding of floating-point data, known as the IEEE-754 standards. For more details on the equations used for floating point representations, see section B4.6 of the Armv8-M Architecture Reference Manual.

The floating-point unit (FPU) is optional in several Cortex-M processors. The version of the Floating-point extension that is supported in the Armv8-M architecture is FPv5. If a floating-point unit (FPU) is available in a the processor, then you can use the floating-point hardware unit to accelerate floating-point operations. When a processor does not support a floating-point unit, then all floating-point calculations are carried out using runtime library functions. This effectively means that floating-point calculations execute more slowly.

Helium technology is the M-Profile Vector Extension (MVE) for the Arm Cortex-M processor series and is an extension of the Armv8.1-M architecture. It delivers a significant performance uplift for machine learning (ML) and digital signal processing (DSP) applications. Helium provides a Single Instruction Multiple Data (SIMD) capability, operating on vectors of elements of the same data type. These data types may be floating-point or integer. Integer elements may be signed or unsigned 8-, 16-, 32-, or 64-bit, while floating-point elements may be single 32-bit or half precision 16-bit. Because the Helium and Floating-point register banks are shared, some Armv8-M Floating-point extension instructions can act on 64-bit wide data, for example `VLDR.64`, and 64-bit double precision floating-point may optionally be supported in hardware.

Access to the floating-point and MVE registers is controlled by the `CP10` and `CP11` bits in the `CPACR`, `NSACR`, and `CPPWR` registers. Subsequent sections in this guide provide more details about these registers.

4.2 Floating-point and MVE registers overview

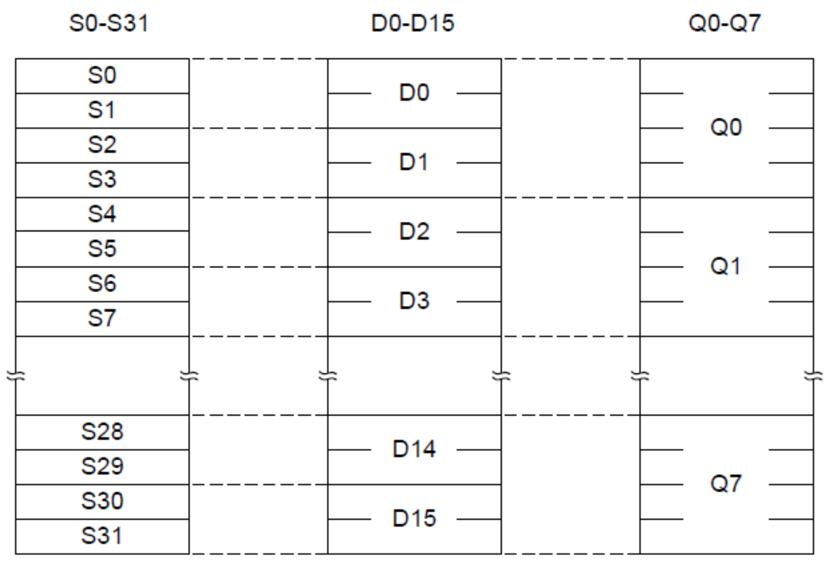
When the optional Floating-point Extension is included, it introduces the following:

- The FP extension registers S0-S31. These FP registers can be viewed as sixteen doubleword registers D0-D15 or eight quadword MVE Vector registers Q0-Q7, as shown in the figure below.
- A new system register, the Floating Point Status and Control Register (FPSCR).

- New memory-mapped registers included in the System Control Block (SCB) memory space, as follows:
  - CPACR, NSACR, CPPWR
  - FPCAR, FPCCR
  - FPDSCR

The floating-point register bank contains thirty-two 32-bit registers which can be viewed as sixteen 64-bit double word registers for double precision floating-point operations.

Figure 4-1: Floating-point register bank



The Arm Architecture Procedure Call Standard specifies that S0-S15 are caller-saved registers. If function A calls function B, then function A must save the contents of these registers before calling function B, because these registers can be changed by the function call. S16-S31 are callee-saved registers. If function A calls function B, and function B needs to use more than 16 registers for its calculations, it must first save the contents of these registers and then restore these registers from the stack before returning to function A.

When the floating-point extension is implemented and enabled in a system, the additional floating-point caller-saved registers S0-S15, FPSCR, and VPR must be saved and restored on exception entry and return. If the exception handler does not need to use the floating-point unit, the Lazy State Preservation feature avoids the timing overhead of saving and restoring the FPU context. See Lazy floating-point state preservation for more information.

Note

The floating-point register bank should be initialized to a known value before using it.

4.2.1 Floating-point Status and Control Register (FPSCR)

The FPSCR holds the arithmetic result flags, sticky status flags, and control bit fields.

The N, Z, C, and V flags are updated by floating-point comparison operations. The results of a floating-point compare for a conditional execution can be performed by copying the FPSCR N, Z, C, and V flags to APSR as follows:

```
VMRS APSR_nzcv,FPSCR      // Copy flags from FPSCR to flags in APSR.
```

For more information about the FPSCR register bit fields, see section D1.2.102 in the Armv8-M Architecture Reference Manual.

4.2.2 Memory-mapped FPU registers

This table summarizes the registers that play a vital role when the Floating-point extension is implemented.

Address	Register	CMSIS-CORE Symbol	Function
`0xE000ED88`	Coprocessor Access Control Register (CPACR)	SCB->CPACR	Global enable and disable of the FPU, coprocessors if implemented, and Arm Custom instructions.
`0xE000ED8C`	Non-secure Access Control Register (NSACR)	SCB->NSACR	When the TrustZone security extension is implemented, the NSACR specifies whether each coprocessor can be accessed from Non-secure state or not. Refer to the Security Extension User Guide for more details.
`0xE000EF34`	Floating-point Context Control Register (FPCCR)	FPU->FPCCR	This register controls the exception handling behavior. The behavior and features controlled by this register include lazy floating-point preservation. If the TrustZone security extension is implemented, this register allows you to control the security settings for handling the floating-point context.
`0xE000EF38`	Floating-point Context Address Register (FPCAR)	FPU->FPCAR	This register stores the address of the space reserved on the stack for the floating point stack frame when the Lazy floating-point state preservation feature is being used. See Lazy floating-point state preservation for more information.
`0xE000EF3C`	Floating-point Default Status Control Register (FPDSCR)	FPU->FPDSCR	Default values of the FPSCR. The FPDSCR value is copied to the FPSCR when creating a new floating-point context.
`0xE000E00C`	Coprocessor Power Control Register (CPPWR)	ICB->CPPWR	Specifies whether coprocessors are permitted to enter a non-retentive power state.

4.2.2.1 CPACR

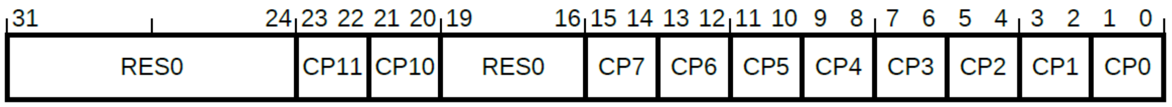
The CPACR register is a part of the System Control Block (SCB).

It allows you to enable or disable the following:

- Floating-point and MVE units
- Coprocessors
- Arm Custom instructions

The CPACR is located at address `0xE000ED88` and is accessed as `SCB->CPACR` in the CMSIS-Core framework. Bits 0-15 are reserved for the coprocessor and Arm Custom instructions. Bits 16-19 and 24-31 are reserved. The following diagram shows he CPACR bit assignments.

Figure 4-2: CPACR bit assignments



The CPACR programmer's model provides bit fields that enable or disable up to 16 coprocessors. The Floating-Point Unit (FPU) and MVE access privileges are defined in coprocessor 10 (CP10) and coprocessor 11 (CP11). To enable the FPU or MVE, both CP10 and CP11 bit fields should have identical values written to CPACR.

For example, before using the FPU, program the CPACR to enable the FPU as follows:

```
SCB->CPACR |= 0x00F00000;    // Enables floating-point unit for full access
```

This step is typically carried out inside the `SystemInit()` function provided in the device software package file. `SystemInit()` is executed by reset handler.

Refer to the synchronous-fault example, together with the example project source code at https://github.com/ARM-software/m-profile-user-guide-examples/tree/main/Exception_model/synchronous-fault, to see how the CPACR register is configured to enable the FPU.

4.2.2.2 NSACR

The NSACR register defines the Non-secure access permissions for the Floating-point extension and coprocessors CP0 to CP7. If the MVE extension is implemented, this register specifies the Non-secure access permissions for MVE.

NSACR is located at address `0xE000ED8C` and can be accessed as `SCB->NSACR` in the CMSIS-Core framework. To enable the FPU to be used by Non-secure software, Secure software needs to set both CP10 and CP11 in the NSACR as follows:

```
SCB->NSACR |= 0x00000C00;   // Enable the floating-point unit for Non-secure use.
```

4.2.2.3 FPCCR

The Floating-Point Context Control Register (FPCCR) controls exception handling behavior. The behavior and features controlled by this register include lazy stacking. If the Security extension is implemented, the FPCCR also specifies the security settings for handling the floating-point context. In most applications, Secure software needs to configure the FPU's security settings in the FPCCR.

For more information about the FPCCR register, see section D1.2.99 of the Armv8-M Architecture Reference Manual.

4.2.2.3.1 FPCCR.ASPEN

The Automatic State Preservation ENable (ASPEN) bit in the FPCCR register enables automatic state preservation and restoration of the floating-point context on exception entry and exception return. By default, the FPCCR.ASPEN bit is set to 1 out of reset.

From Armv8.1-M onwards, clearing the FPCCR.ASPEN bit to zero by software is deprecated.

4.2.2.3.2 FPCCR.LSPEN

The Lazy State Preservation Enable bit in the FPCCR register enables lazy state preservation. The lazy stacking feature is enabled when both the FPCCR.LSPEN and FPCCR.ASPEN bits are set to 1. By default, both FPCCR.LSPEN and FPCCR.ASPEN bits are set to 1 out of reset. Therefore software developers do not need to configure the ASPEN and LSPEN bits to enable the Lazy floating-point state preservation feature. The CONTROL.FPCA bit is automatically set to 1 when a floating-point instruction is executed when the FPU is enabled using CPACR.

4.2.2.3.3 FPCCR.CLRONRET

When the FPCCR.CLRONRET, CLeaR ON RETurn, bit is set to 1, it clears the floating-point registers S0-S15, FPSCR, and VPR on exception return.

4.2.2.3.4 FPCCR.LSPACT

When LSPACT is set to 1, floating-point context saving on the reserved stack space is deferred. The address of this reserved stack frame is captured in the FPCAR register. If a floating-point instruction is executed when LSPACT is 1, then a lazy stacking process for the floating-point registers is activated. Once all floating-point registers are saved on to the stack, the LSPACT bit is set to zero.

4.2.2.4 FPCAR

When an exception occurs with the current context having active floating-point context, then the exception stack frame needs to save registers from both of the following:

- the integer register bank: R0-R3, R12, LR, ReturnAddress, and RETPSR
- the FPU register bank: S0-S15, FPSCR, and VPR).

To reduce interrupt latency, by default lazy stacking is enabled. Lazy stacking means that the stacking mechanism reserves the stack space for floating-point registers, but does not actually push floating-point registers onto the stack frame until it executes a floating-point instruction inside the exception handler.

The FPCAR register is part of this lazy stacking mechanism. The FPCAR register points to a section of stack space within the exception stack frame that has been reserved for storing the floating-point registers S0-S15, FPSCR, and VPR. The address value in the FPCAR is automatically generated by the hardware of the processor. See Lazy floating-point state preservation for more details.

4.2.2.5 FPDSCR

The FPDSCR register is a memory-mapped register that holds default configuration information such as rounding modes for floating-point status control data. The FPDSCR value is copied to the FPSCR when a new floating-point context is created. The FPDSCR defines the FPU configuration when the exception handlers start, including the OS kernel as most parts of the OS are executing in Handler mode.

4.3 Floating-point context handling mechanisms

When developing Cortex-M software, creating an exception handler is quite simple. For example, you can declare a timer handler as follows:

```
void Timer0_Handler(void)
{
   ... // Status checks
   ... // Clear interrupt request
   return;
}
```

The function name `Timer0_Handler` in this example needs to match the handler name declared in the vector table, which itself is configured in device-specific startup code.

When the Floating-point extension is implemented and enabled in a system, on exception entry and return, the additional floating-point caller-saved registers S0-S15 and FPSCR must be saved on to the stack frame. If the MVE extension is implemented, then the VPR register is also saved and restored automatically during exception entry and return sequences. Along with the floating-point register bank, the memory-mapped registers and controls described in the following sections play a vital role in exception handling. See Memory-mapped FPU registers for more information.

4.3.1 CONTROL.FPCA

CONTROL.FPCA plays a crucial role in deciding whether a floating-point context is active or not-active. A floating-point context includes the registers S0-S15, FPSCR, and VPR. CONTROL.FPCA indicates whether the current context has floating-point state. This bit is:

- Set to 1 when a processor executes a floating-point instruction if FPCCR.ASPEN is set
- Cleared to zero when entering an exception handler, that is, when a new context is started
- Saved and restored during exception handling in EXC_RETURN[4]
- Cleared to zero out of reset

The FPCA bit is available only when floating-point or MVE extension is implemented. The CONTROL Register is not memory mapped. In general, CONTROL.FPCA is handled automatically by the processor and it does not need to be modified by software.

4.3.2 EXC_RETURN

Bit 4 of EXC_RETURN is set to 0 at exception entry if the pre-empted task has a floating-point context, that is when CONTROL.FPCA is 1.

When EXC_RETURN[4] is 0, it indicates that the longer stack frame is used for stacking, that is R0-R3, R12, LR, ReturnAddress, RETPSR, S0-S15, FPSCR, and VPR).

When EXC_RETURN[4] is 1, then it indicates that the shorter stack frame is used for stacking.

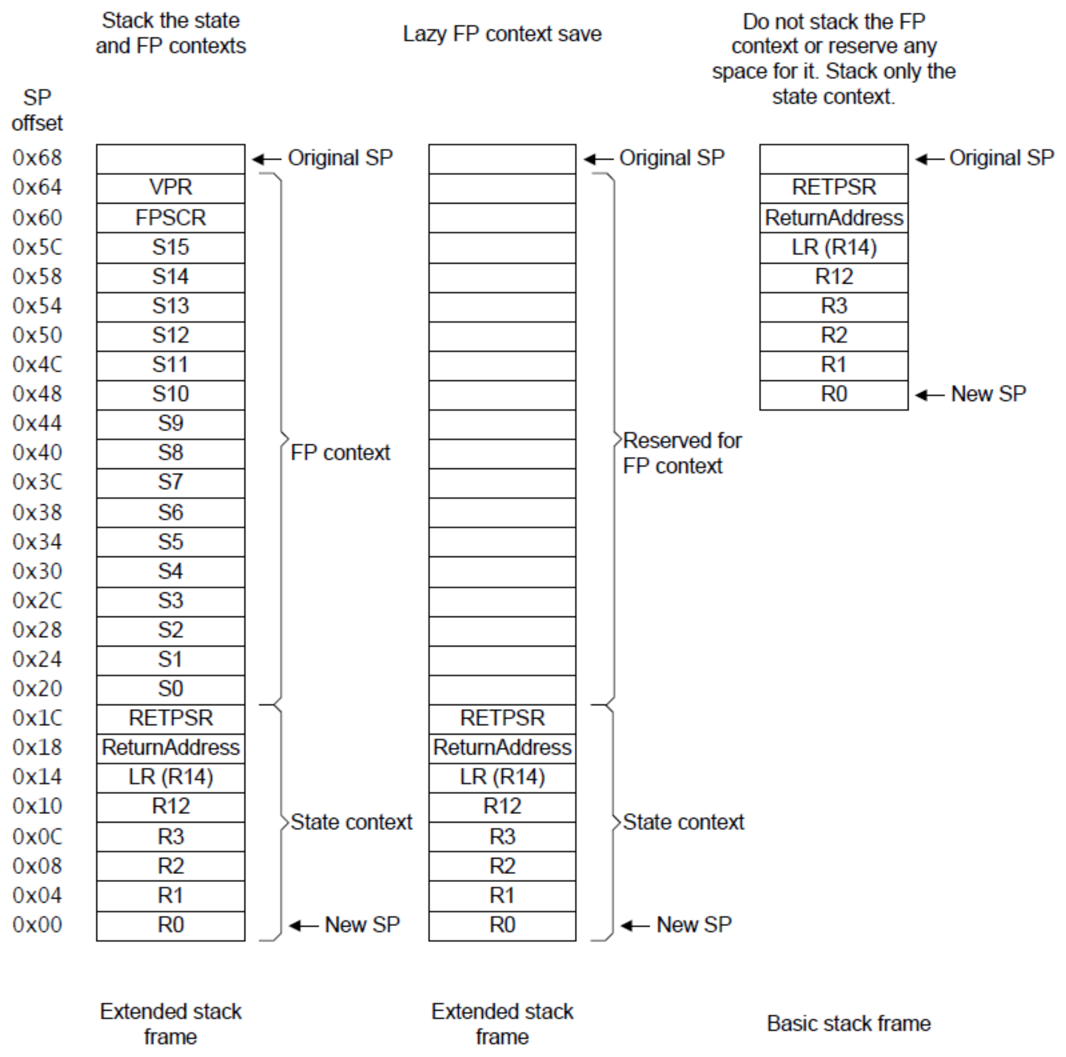
Note

EXC_RETURN (Exception Return) is a code value generated automatically by Cortex-M processors when entering an exception handler. The value is stored in the Link Register (LR) and is used at exception return. Several bits of this code value are used to store information about the status of the processor before the exception, for example, which stack pointer was being used.

4.3.3 Exception stack frame format

The exception stack frame has an additional floating-point context format type shown in the following diagram. This additional format type permits the saving of registers S0-S15, FPSCR, and VPR in the FPU.

Figure 4-3: Exception stack frame format, with and without the floating-point context



The Arm Architecture Procedure Call Standard specifies that a C function must preserve registers S16-S31. The other registers in the FPU, that is, S0-S15, FPSCR, and VPR are always saved automatically. These registers can also be modified by a C function.

To allow an exception handler to be written as a normal C function, the processor must automatically save the S0-S15 registers, FPSCR, and VPR on the stack. There are four possible outcomes of saving the floating-point (FP) context onto the stack pointer of the background context on an exception entry:

1. Do not stack any FP context at all.

   If the FP/MVE registers are not used, indicated by 0 in the FPCA bit in the CONTROL register, only integer registers are saved. This is shown in the basic stack frame in the figure above, and described in Stack frames.
2. Extended Stack frame.

   This stack frame preserves both integer and floating-point registers as specified in the AAPCS standard on every exception entry. This is shown in the extended stack frame in the figure above, and described in Extended stack frame, automatic stacking.
3. Lazy floating-point state preservation.

   An empty space is reserved onto the stack frame on an exception entry. The stack frame is populated with FP registers only when a floating-point instruction is executed in the exception handler or Interrupt Service Routine (ISR). See Lazy floating-point state preservation for more information.
4. Save the entire floating-point register bank and FPSCR.

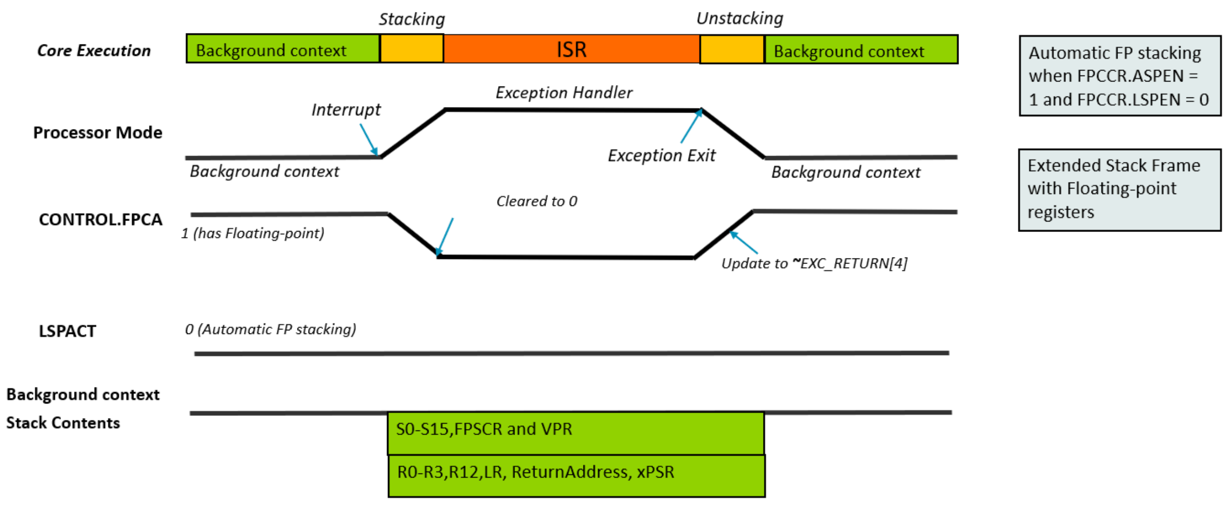
   In this situation, the S0-S31, FPSCR, and VPR registers are saved to the stack on an exception entry. This type of stacking occurs when an exception is taken to Non-secure from Secure state. Refer to the Security Extension User Guide for more details.

4.3.4 Extended stack frame, automatic stacking

When the floating-point extension is enabled and if there is a floating-point context, then there is an optional feature to save floating-point caller saved registers onto the stack frame along with other registers. This feature is called automatic stacking with floating-point context. Automatic stacking is enabled when FPCCR.ASPEN is set to 1, and FPCCR.LSPEN is set to 0. Automatic stacking saves the following registers for each exception entry, and restores them back on exception return:

- S0-S15
- FPSCR
- VPR
- R0-R3
- R12
- LR
- PC
- REPSR

Figure 4-4: Automatic Stacking



If the CONTROL.FPCA bit is set and the automatic state saving feature has been enabled, the exception stack frame with floating-point storage is used. This is shown as the extended stack frame in the diagram in Exception stack frame format. This is because the register values in the FPU might be required by the current context after exception handling is complete.

The stacking of floating-point registers has the following effects:

- Increased stack frame size
- Potentially increased interrupt latency in interrupt processing
- Increased context switching time in an embedded OS

For exception handling in applications without an OS, the automatic hardware state preservation is sufficient and is easy to use. You can write exception handlers as normal C functions, and the automatic stacking mechanism handles the required floating-point register stacking and unstacking if required.

For developers of an embedded OS, the situation is more complex. To permit a multi-tasking system to use the FPU or MVE in multiple tasks, you must update the OS or Real-Time Operating System (RTOS) to handle context saving of the extra registers, S16-S31.

During context switching, the OS must:

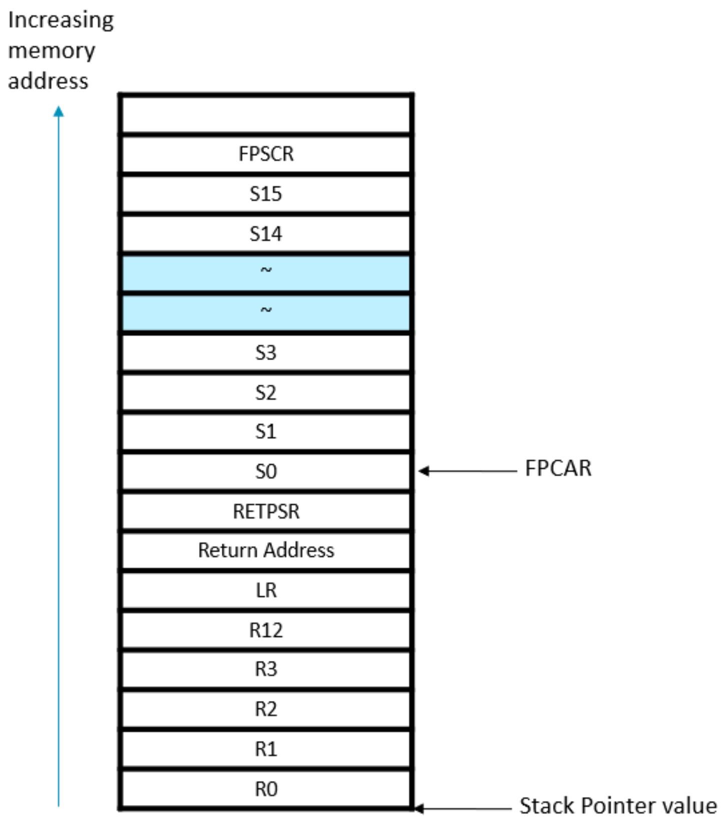
1. Determine whether an application task has used the FP or MVE registers, using bit 4 of EXC_RETURN. If EXC_RETURN[4] is 0, then it indicates that the FP or MVE registers are used.
2. Save the floating-point context if required.
3. Restore the floating-point context for the next task if required.
4. Switch to the next task using an exception return, with the EXC_RETURN code value matching the stack frame type.

4.3.5 Lazy floating-point state preservation

In order to reduce interrupt latency, Cortex-M processors with a floating-point or MVE unit have a feature called lazy stacking. As described in Stack frames), when we need to stack the required floating-point registers for each exception, additional memory pushes for the FP and MVE registers would be required each time an exception occurred.

When an application has previously used the FP or MVE registers, the CONTROL.FPCA bit is automatically set to 1. If an exception occurs, and if the lazy stacking feature is enabled, the processor reserves extra space in the stack frame for the S0-S15 registers and FPSCR. However, the actual stacking of these registers does not take place, and the Lazy State Preservation Active (LSPACT) bit in FPCCR is set to 1.

Figure 4-5: Extended stack frame with floating-point context



Bit[4] of the EXC_RETURN value, generated at the exception entry, is set to 0 to indicate that the exception stack frame contains stack space for the floating-point registers, although the actual register contents are not present. The Floating-Point Context Address Register (FPCAR) is set to store the address of the reserved stack space for the floating-point register.

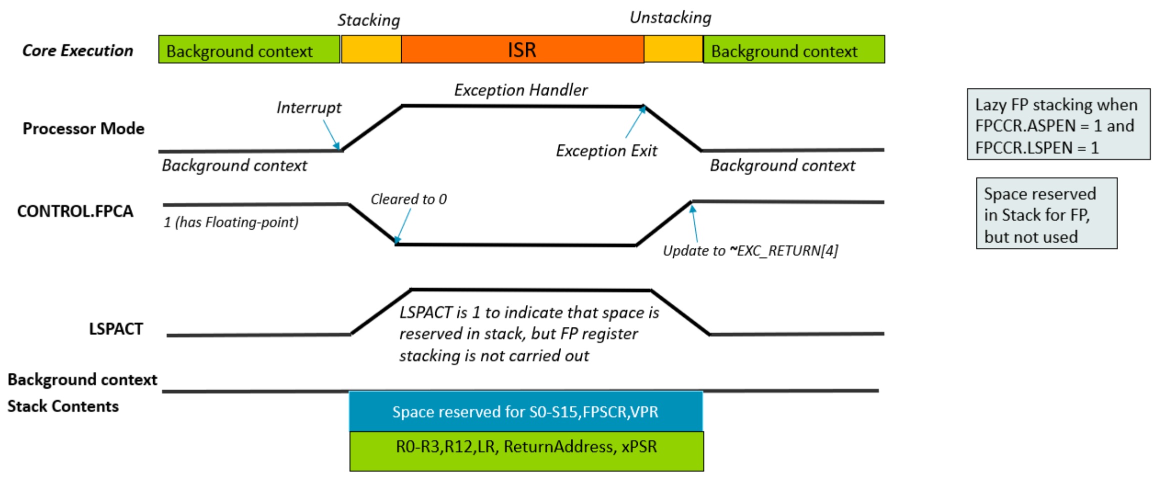
There are two possible scenarios possible when the lazy floating-point state preservation feature is enabled:

1. Lazy floating-point state preservation is invoked, but not used.

If the exception handler does not use the FP or MVE unit, then LSPACT stays HIGH until the end of the exception. When returning from the exception, the processor hardware detects that bit[4] of EXC_RETURN is 0 and LSPACT is 1. This indicates that the stack frame contains space for the floating-point registers, but they were not pushed onto the stack. This means that the unstacking of the floating-point registers can be skipped, as the registers still contain the values belonging to the background context being returned to.

The following diagram shows that, at exception return, although EXC_RETURN[4] is 0, LSPACT is 1. This indicates that the floating-point registers were not pushed to the stack. S0-S15, FPSCR, and VPR are not unstacked and remain unchanged.

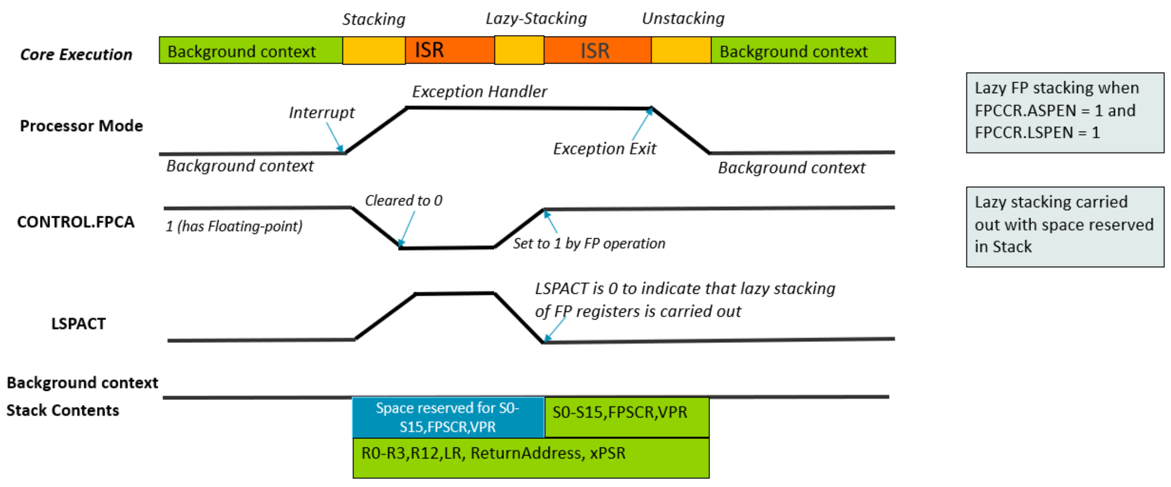
Figure 4-6: Lazy state preservation invoked, but not used


Stacking
Unstacking
Core Execution
Background context
ISR
Background context
Exception Handler
Lazy FP stacking when FPCCR.ASPEN = 1 and FPCCR.LSPEN = 1
Interrupt
Processor Mode
Exception Exit
Background context
Background context
Space reserved in Stack for FP, but not used
CONTROL.FPCA
Cleared to 0
1 (has Floating-point)
Update to ~EXC_RETURN[4]
LSPACT
LSPACT is 1 to indicate that space is reserved in stack, but FP register stacking is not carried out
Background context
Stack Contents
Space reserved for S0-S15,FPSCR,VPR
R0-R3,R12,LR, ReturnAddress, xPSR

2. Lazy floating-point state preservation invoked and used by the exception handler.

   If the exception handler uses the FP or MVE unit, when the first floating-point instruction is executed the processor pushes the floating-point registers on to the stack space reserved earlier during exception entry, and LSPACT is cleared. The stacked registers are S0-S15, FPSCR, and VPR if the Armv8.1-M MVE extension is implemented.

Figure 4-7: Lazy state preservation invoked and used in exception handler


Stacking
Lazy-Stacking
Unstacking
Core Execution
Background context
ISR
ISR
Background context
Lazy FP stacking when FPCCR.ASPEN = 1 and FPCCR.LSPEN = 1
Processor Mode
Interrupt
Exception Handler
Exception Exit
Background context
Background context
CONTROL.FPCA
1 (has Floating-point)
Cleared to 0
Set to 1 by FP operation
Lazy stacking carried out with space reserved in Stack
LSPACT
LSPACT is 0 to indicate that lazy stacking of FP registers is carried out
Background context Stack Contents
Space reserved for S0-S15,FPSCR,VPR
S0-S15,FPSCR,VPR
R0-R3,R12,LR, ReturnAddress, xPSR

The program execution then continues. As shown in the figure above, a floating-point instruction is executed during the ISR execution and triggers the deferred stacking. During this operation, the address of the reserved stack space stored in FPCAR is used to stack the FP and MVE registers during lazy stacking. At the end of the exception handler, the processor hardware detects that EXC_RETURN[4] is 0 and LSPACT is 0, indicating that the stack frame contains pushed floating-point registers and unstacks them accordingly. If FPCCR.CLRONRET is set to 1, then the floating-point registers, S0-S15, FPSCR, and VPR, are set to zero on exception return.

The FPCCR.CLRONRET bit is particularly useful because the contents of the FP and MVE register bank are not exposed when switching between different privileged states.

Refer to the context-switch-fp example to see how lazy context preservation is used in typical RTOS context-switching operations.

5. Use case examples

This chapter shows several use case examples related to exception handling settings.

- irq-priority-basic

  A basic example that shows how to relocate the vector table, pre-emption and tail-chaining behavior between different IRQs with different priorities, and the effect of group priority and sub priority on these behaviors.
- priority-boost-types

  An example to generate situations where the current priority level can be boosted using BASEPRI.
- system-exceptions

  An example that shows how to use the SysTick and PendSV built-in system exceptions.
- svc-number-as-parameter

  A simple example that shows how to select different functions by extracting and switching SVC number.
- synchronous-fault

  A simple example that shows the process of triggering a UsageFault and fixing it in its handler.
- interrupt-deprivileging

  An advanced example that shows how to forward and deprivilege an interrupt from a privileged background to the unprivileged thread.
- context-switch-fp

  An advanced example that shows context switching between tasks when enabling the FPU.

The source code for these examples can be found in the GitHub repository.

5.1 General information

For most basic applications, programs can be completely written in C language. The C compiler compiles the C program code into object files and then generates the executable program image file using the linker.

5.1.1 What does the program image contain?

When a project is built using the toolchain, it generates a program image. Inside this program image, in addition to the application code that we would want to run, there is also a range of other software components. The program image contains the following:

- Vector table
- Reset handler or startup code
- C startup code
- C runtime library functions
- Application code

The following sections provide more information about each of these software components.

5.1.1.1 Vector table

In Arm Cortex-M processors, the vector table contains the starting addresses of each exception and interrupt. One of the exceptions is reset. After reset, the processor fetches the reset vector, the starting address of the reset handler, from the vector table and starts executing the reset handler. The first word in the vector table defines the starting value of the Main Stack Pointer (MSP). If the vector table is not configured correctly in the program image, the device cannot start.

In the Arm Development Studio project examples shown in this guide, the vector table is defined by device-specific startup code in `<example_project>/RTE/Device/ARMv8MML/startup_ARMv8MML.c`. Here is a snippet of the vector table from the startup code.

```
const VECTOR_TABLE_Type __VECTOR_TABLE[48] __VECTOR_TABLE_ATTRIBUTE = {
  (VECTOR_TABLE_Type)(&__INITIAL_SP),      /*     Initial Stack Pointer */
  Reset_Handler,                           /*     Reset Handler */
  NMI_Handler,                             /*     NMI Handler */
  HardFault_Handler,                       /*     Hard Fault Handler */
  MemManage_Handler,                       /*     MPU Fault Handler */
  BusFault_Handler,                        /*     Bus Fault Handler */
  UsageFault_Handler,                      /*     Usage Fault Handler */
  SecureFault_Handler,                     /*     Secure Fault Handler */
  0,                                       /*     Reserved */
  0,                                       /*     Reserved */
  0,                                       /*     Reserved */
  SVC_Handler,                             /*     SVC Handler */
  DebugMon_Handler,                        /*     Debug Monitor Handler */
  0,                                       /*     Reserved */
  PendSV_Handler,                          /*     PendSV Handler */
  SysTick_Handler,                         /*     SysTick Handler */

  /* Interrupts */
  Interrupt0_Handler,                      /*     Interrupt 0 */
  Interrupt1_Handler,                      /*     Interrupt 1 */
  Interrupt2_Handler,                      /*     Interrupt 2 */
  Interrupt3_Handler,                      /*     Interrupt 3 */
  Interrupt4_Handler,                      /*     Interrupt 4 */
  Interrupt5_Handler,                      /*     Interrupt 5 */
  Interrupt6_Handler,                      /*     Interrupt 6 */
  Interrupt7_Handler,                      /*     Interrupt 7 */
  Interrupt8_Handler,                      /*     Interrupt 8 */
  Interrupt9_Handler                       /*     Interrupt 9 */
```

```
    ...
};
```

This vector table uses the following symbols:

- __VECTOR_TABLE

  This symbol name defines the static interrupt vector table. The name must comply with any compiler and linker conventions, for example if used for vector table relocation. CMSIS-Core specifies a common default for supported compilers.

- __VECTOR_TABLE_ATTRIBUTE

  This symbol name defines the additional declaration specifications to be used when defining the static interrupt vector table.

  Both __VECTOR_TABLE and __VECTOR_TABLE_ATTRIBUTE are expected to be used only by the startup file `<example_project>/RTE/Device/ARMv8MML/startup_ARMv8MML.c`.

- __INITIAL_SP

  The initial stack pointer value is defined in CMSIS Pack files, for example `cmsis_armclang.h` or `cmsis_gcc.h`.

5.1.1.2 Reset_Handler()

The reset handler or startup code is the first piece of software executed after a system reset. Typically, it is used for setting up configuration data for the C startup code such as the address range for stack and heap memories, and then branches into the C startup code. It is considered best practice to initialize stack pointers with their limits before entering C startup code. Because this project uses the CMSIS-CORE framework, the reset handler executes the `SystemInit()` function which sets up the configuration for clocks and PLLs, before branching to C startup code.

```
    __NO_RETURN void Reset_Handler(void)
    {
      __set_PSP((uint32_t)(&__INITIAL_SP));

      __set_MSPLIM((uint32_t)(&__STACK_LIMIT));
      __set_PSPLIM((uint32_t)(&__STACK_LIMIT));

    #if defined (__ARM_FEATURE_CMSE) && (__ARM_FEATURE_CMSE == 3U)
      __TZ_set_STACKSEAL_S((uint32_t *)(&__STACK_SEAL));
    #endif

      SystemInit();                          /* CMSIS System Initialization */
      __PROGRAM_START();                     /* Enter PreMain (C library entry
point) */
    }
```

Note

In a CMSIS project, the PSP is initialized in the same region as MSP by default. However, if the PSP is ever used it will cause data corruption. If the PSP is required, it must be moved into another region by changing the scatter file.

Note

Depending on the development tools being used, the reset handler can be optional. If the reset handler is omitted, the C startup code is executed directly instead. The startup code is typically provided by microcontroller vendors and is also often bundled inside toolchains. They can be either in the form of assembly or C code.

5.1.1.3 Default_Handler()

The device-specific startup code in `<example_project>/RTE/Device/ARMv8MML/startup_ARMv8MML.c` provides weak aliases for each exception handler to `Default_Handler`.

Note

The weak symbol is a special linker symbol that denotes a function that can be overridden during link time.

Because the exception handlers are marked as weak aliases, any function with the same name will override this definition. This allows the programmer to define their own handlers without the need to change the device-specific startup code.

For example, consider the following snippet of the startup code:

```
void Interrupt0_Handler     (void) __attribute__ ((weak, alias("Default_Handler")));
```

This code tells the linker to assign the `Default_Handler` to `Interrupt0_Handler` if the programmer does not provide an `Interrupt0_Handler` function themselves.

In the rest of the start-up code there is similar code for each interrupt handler. This allows the code to create a default handler without requiring the programmer to explicitly assign a specific handler for each interrupt.

5.1.1.4 C startup code

When using high level languages like C/C++, the processor needs to execute a piece of program code to configure the program execution environment. This configuration includes:

- Setting initial data values in SRAM, for global variables for example.
- Zero initialization of data memory for variables that are uninitialized at load time.
- Initializing the data variables controlling heap memory, for `malloc()` for example.

After initialization, the C startup code branches to the start of the `main()` program. The C startup code is automatically inserted by the toolchain. It is toolchain-specific and is not inserted by the toolchain if the program is written in assembly.

Note

For Arm compilers, the C startup code is labeled as `__main`, while the startup code generated by GNU C compilers is normally labeled as `_start`.

5.1.1.5 C runtime library functions

C library code is inserted into the program image by the linker when particular C/C++ functions are used. C library code can also be included for specific data processing tasks, for example floating-point calculations.

5.1.1.6 Relocating the vector table

In most of the example cases, because we read the IRQ state in each handler, we use a general handler function rather than overwriting each instance of handler code. When entering `main()`, the program calls the `Vector_Table_Relocation()` function to relocate the vector table and have all IRQ handlers call the same handling function in the vector table.

The vector table starts at `0x0` by default. It can be relocated to a new memory location by configuring the Vector Table Offset Register (VTOR). We create an array to copy the content of the original vector table and then set the value of VTOR to the address of this array.

The following code shows how to relocate the vector table:

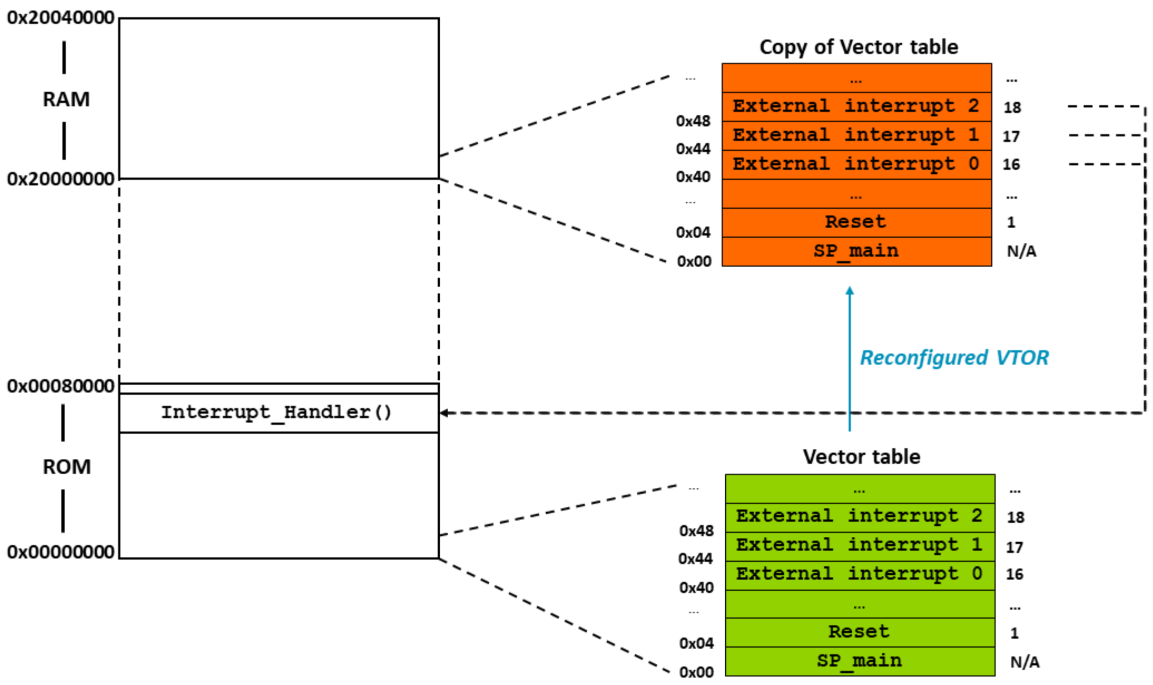
```
    <...>
    /* Parameters for new vector table */
    #define VECTORTABLE_SIZE                48
    #define VECTORTABLE_ALIGNMENT           0x100U

    extern uint32_t __VECTOR_TABLE[VECTORTABLE_SIZE];
    uint32_t new_vectorTable[VECTORTABLE_SIZE] __attribute__((aligned
 (VECTORTABLE_ALIGNMENT)));

    void Vector_Table_Relocation(void){
      /* Copy the original handler address into new vector table */
      memcpy(new_vectorTable, __VECTOR_TABLE, sizeof(uint32_t));

      /* Replace the element with new handler address */
      new_vectorTable[Interrupt0_IRQn + IRQ_offset] = (uint32_t)Interrupt_Handler;
      new_vectorTable[Interrupt1_IRQn + IRQ_offset] = (uint32_t)Interrupt_Handler;
      new_vectorTable[Interrupt2_IRQn + IRQ_offset] = (uint32_t)Interrupt_Handler;

      /* Get information about vector table */
      printf("Vector table address is 0x%08x\n", SCB->VTOR);
      printf("The IRQ0's vector address is 0x%08x\n",
            (uint32_t)&__VECTOR_TABLE[Interrupt0_IRQn+IRQ_offset]);

      /* Change the VTOR into new vector table address */
      SCB->VTOR = (uint32_t)&new_vectorTable;
      __DSB();
      __ISB();

      printf("New vector table address is 0x%08x\n", SCB->VTOR );
      printf("The new IRQ0's vector address is 0x%08x\n",
            (uint32_t)&new_vectorTable[Interrupt0_IRQn+IRQ_offset]);
    }

    <...>
```

This code performs the following operations:

1. Define the new vector table with the given symbol `_VECTOR_TABLE`.
2. Define the `_VECTORTABLE_SIZE` macro. This specifies the number of general purpose interrupt lines. The default value is 496, which can be found in `<example_project>/RTE/Device/ARMv8MML/startup_ARMv8MML.c`. For our example, we do not need to use so many exceptions, so we decrease it to a lower number 48.
3. Define the `_VECTORTABLE_ALIGNMENT` macro. This ensures that vector table is aligned to a power of 2, such that the maximum exception number fits in the vector table. We specified 48 interrupts, so the alignment must be on a 64-word boundary because the table size is 64 words.
4. Create an array containing vector table entries for the new interrupt handlers with the address of function `Interrupt_Handler()`.
5. Configure VTOR with the address of the array.
6. Create `DSB` and `ISB` barriers. The `DSB` barrier is used to guarantee that the register writes complete, while the `ISB` barrier is used to make sure the updates take effect before the execution of next instruction.

An execution flow chart for this code is shown below:

Figure 5-1: Execution flow chart for Vector_Table_Relocation

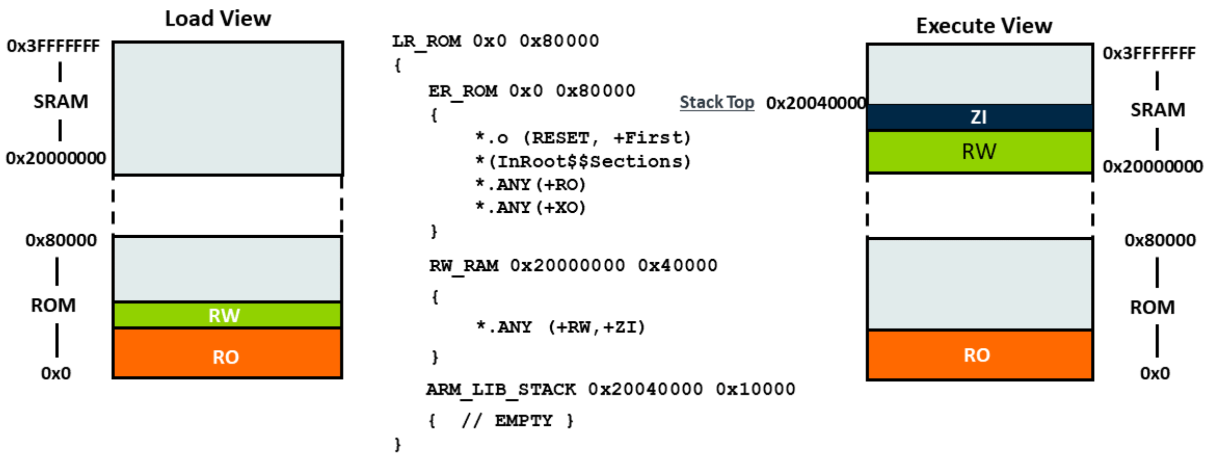


5.1.2 Memory map and scatter file definition

The example projects use the MPS2 Fixed Virtual Platform (FVP) to run the program. The memory map for the MPS2 FVP can be found in MPS2 - memory map for models with the Arm®v8 M additions.

A scatter file gives you the ability to control where the linker places different parts of your image for your target including the location and size of various memory regions that are mapped to ROM, RAM, and FLASH. Considering the target memory map of MPS2 FVP model, we need ROM, RAM, and STACK regions which are provided by default. The following execution regions are defined for this project:

Figure 5-2: Scatter file layout



The following snippet of the scatter file contains the definitions for this project. See the scatter file on GitHub for the full description of the different regions.

You must ensure that sufficient memory space is allocated for stack operations to support the different exceptions. With the Arm Compiler toolchain, memory space for the stack is defined using the ARM_LIB_STACK region. This is defined in the scatter file as follows:

```
<...>

#define __ROM_BASE      0x00000000
#define __ROM_SIZE      0x00080000

#define __RAM_BASE      0x20000000
#define __RAM_SIZE      0x00040000

#define __STACK_SIZE    0x00010000

/*----------------------------------------------------------------------------
  Scatter Region definition
 *----------------------------------------------------------------------------*/
LR_ROM __RO_BASE __RO_SIZE  {                ; load region size_region
  ER_ROM __RO_BASE __RO_SIZE  {              ; load address = execution address
   *.o (RESET, +First)
```

```
        *(InRoot$$Sections)
        .ANY (+RO)
        .ANY (+XO)
       }

       RW_RAM __RW_BASE __RW_SIZE  {
        ; RW data
        .ANY (+RW +ZI)
       }

      ; ============================================================
      ; ARM_LIB_STACK 0x2004_0000 EMPTY -0x10000  ; Stack growing down
      ; ============================================================
       ARM_LIB_STACK __STACK_TOP EMPTY -__STACK_SIZE {   ; Reserve empty region for
stack
       }
   <...>
```

In most of the examples, because the Floating-point extension and the Security extension are not enabled, only 8 words are needed for an exception stack frame, that is an integer stack frame.

Note

The stack pointer memory size must be carefully chosen for the maximum possible number of nested interrupts. If the number of possible nested interrupts is greater, then more stack is needed. A large number of nested interrupts also increases the overall number of cycles to include multiple stacking and unstacking processes, whereas that is not the case with tail-chaining. If the programmer carefully chooses the priority settings necessary for IRQs and exceptions, execution can be optimized.

For additional information about scatter file definitions, please see the following resources:

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

5.1.3 Tool versions

This example project is created, built, and run using following tool versions:

- Arm Development Studio 2022.2
- Arm Compiler for Embedded 6
- Fast Models Fixed Virtual Platforms (FVP) 11.18
- CMSIS 5.9.0, available from the ARM-software/CMSIS_5.

5.2 irq-priority-basic

This example is a basic example that shows how to relocate the vector table, pre-emption and tail-chaining behavior between different IRQs with different priorities, and the effect of group priority and sub priority on these behaviors.

This example provides a basic demonstration of interrupt configuration for Cortex-M processors. The example covers the following areas:

- Nested exceptions
- Pre-emption and tail-chaining behavior
- Configuring Stack pointers, Stack limit registers
- Purpose of the vector table and how to relocate it
- Setting up priority and meaning of group priority and sub-priority

The source code for this example is available at Exception_model/irq-priority-basic.

5.2.1 Project structure

The file structure of the example project is as follows:

```
│    main.c
│    IRQConfig.h
│    IRQConfig.c
└───RTE
    │    RTE_Components.h
    └───Device
        └───ARMv8MML
                |   ARMv8MML_ac6.sct
                |   startup_ARMv8MML.c
                └───system_ARMv8MML.c
```

The files in the example project are as follows:

- `main.c`: The code in this file includes 3 case sections, as follows:
  1. The first case shows pre-emption between nested IRQs.
  2. The second case shows tail-chaining when several IRQs occur at once.
  3. The third case shows the effect on pre-emption and tail-chaining when configuring the group and sub-priority.
- `IRQConfig.c`: The code in this file does the following:
  1. Overwrites the interrupt handlers by relocating the vector table and replaces the element with the new handler.
  2. Sets the priority group, pre-empt priority, and sub-priority.
  3. Prints the interrupt's active and pending status.
- `IRQConfig.h`: Contains macro and function declarations.

- `RTE/Device/ARMv8MML/startup_ARMv8MML.c`: Configures the vector table, then initializes the MSP and PSP.
- `RTE/Device/ARMv8MML/ARMv8MML_ac6.sct`: Scatter file.
- `RTE/Device/ARMv8MML/system_ARMv8MML.c`: Target definitions.

5.2.2 Global IRQ configuration

Before generating and handling IRQs, we need to know which features we want to test. Then, according to the testing scenarios, we perform global IRQ configuration such as setting the priority of IRQs and determining the order in which IRQs are triggered.

5.2.2.1 Trigger IRQs with different scenarios

This use case example demonstrates many different orderings of interrupts. In real life, the interrupts would be generated by external peripherals. This example uses an `IRQPendRequests` array to simulate the arrival of different interrupts. The `IRQPendRequests` array contains details about which interrupts occur at different points in time. Each element of the array corresponds to an invocation of an interrupt handler, with the 32 bits within a given element corresponding to the interrupts to pend from within that interrupt handler. All the interrupts that should be pended from within a given interrupt handler can be pended by copying the value of the element into the Interrupt Set Pending Register, NVIC_ISPR. For instance, setting `SETPEND[m]` in `NVIC_ISPRn` indicates that interrupt 32n+m is pending.

For example, the following code triggers IRQ1 with the IRQ0 handler, and triggers IRQ2 with the IRQ1 handler:

```
/* Parameters for pending different interrupts at each IRQ handler */
#define IRQ_PENDNum                      3 /* Number of IRQs to trigger */

#define InIRQ0Handler                    0 /* Index for pending request array */
#define InIRQ1Handler                    1
#define InIRQ2Handler                    2

<...>
uint32_t IRQPendRequests[IRQ_PENDNum];

/* Pend IRQ1 at IRQ0 handler, pend IRQ2 at IRQ1 handler */
IRQPendRequests[InIRQ0Handler] = 1 << (uint32_t)Interrupt1_IRQn;
IRQPendRequests[InIRQ1Handler] = 1 << (uint32_t)Interrupt2_IRQn;
<...>
```

The `Interrupt_Handler()` function receives the exception number of the current active interrupt, reads the related element of this array, and then sets the element to ISPR. By reading the ISPR, we can output the pending and active flags as follows:

```
<...>

void Interrupt_Handler(void) {
  /* Get the current active exception number */
  uint32_t IRQNum = __get_IPSR() - IRQ_offset;
  uint32_t IRQsToPend = IRQPendRequests[IRQNum];
  printf("We are in IRQ %d Handler!\n", IRQNum);
  Print_PendIRQ(IRQsToPend);

  /* Force the new pended interrupts to be recognized */
  NVIC->ISPR[0] = IRQsToPend;

  __DSB();
  __ISB();

  Print_PendAndActiveStatus();
}

<...>
```

5.2.3 Case 1: IRQ pre-emption

When the processor is executing an exception handler, an exception can pre-empt the exception handler if its priority is higher than the priority of the exception being handled. In this case, processor pre-emption is configured by setting the priority levels as `IRQ2 > IRQ1 > IRQ0`. Note that a higher priority level is a lower numeric value.

Considering the effect of `NVIC_PRIO_BITS`, that is only the lowest 3 bits are valid, the code sets the priorities for IRQs as follows:

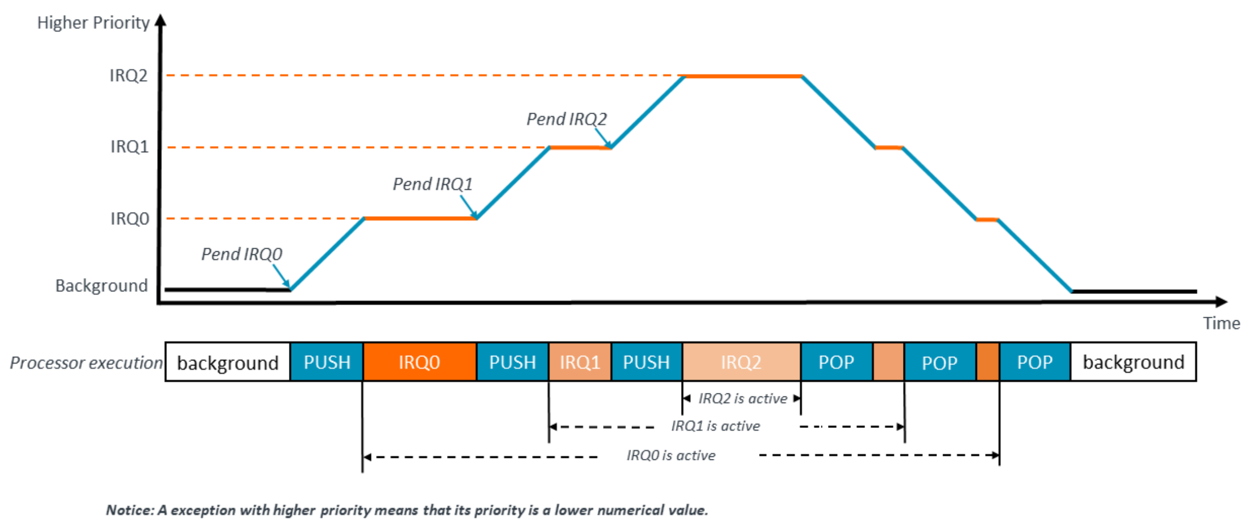
```
<...>

NVIC_SetPriority(Interrupt0_IRQn, 0xFF);
NVIC_SetPriority(Interrupt1_IRQn, 0x66);
NVIC_SetPriority(Interrupt2_IRQn, 0x44);

<...>
```

The code then sets `IRQPendRequests` to pend IRQ1 at the IRQ0 handler and pend IRQ2 at the IRQ1 handler. Finally, the code triggers IRQ0. Executing the code shows pre-emption taking place as each interrupt pre-empts its predecessor.

```
<...>

/* Pend IRQ1 at IRQ0 handler, pend IRQ2 at IRQ1 handler */
IRQPendRequests[InIRQ0Handler] = 1 << (uint32_t)Interrupt1_IRQn;
IRQPendRequests[InIRQ1Handler] = 1 << (uint32_t)Interrupt2_IRQn;

/* Step3: Trigger IRQ0 firstly */
NVIC_SetPendingIRQ(Interrupt0_IRQn);
printf("Case:1 is completed! \n");

<...>
```

The following flowchart shows how this code executes:

Figure 5-3: Execution flow chart for IRQ pre-emption example



An exception with a higher priority means that its priority is a lower numerical value.

5.2.3.1 Output in target console

The code reads SCB->ICSR to get the status of each IRQ. The output shows that IRQ2 is handled first, then IRQ1, and finally IRQ0.

```
We are in IRQ 0 Handler!
Setting IRQ 1 to pend
We are in IRQ 1 Handler!
Setting IRQ 2 to pend
We are in IRQ 2 Handler!
There is more than one active exception.
The number of the highest priority active exception is 18
There is more than one active exception.
The number of the highest priority active exception is 17
There is only one active exception.
The number of the highest priority active exception is 16
Case:1 is completed!
```

5.2.4 Case 2: tail-chaining

A different priority configuration can cause exceptions to be pended rather than taken straight away. Tail-chaining is used to optimize the back-to-back processing of exceptions without the overhead of state saving and restoration between interrupts.

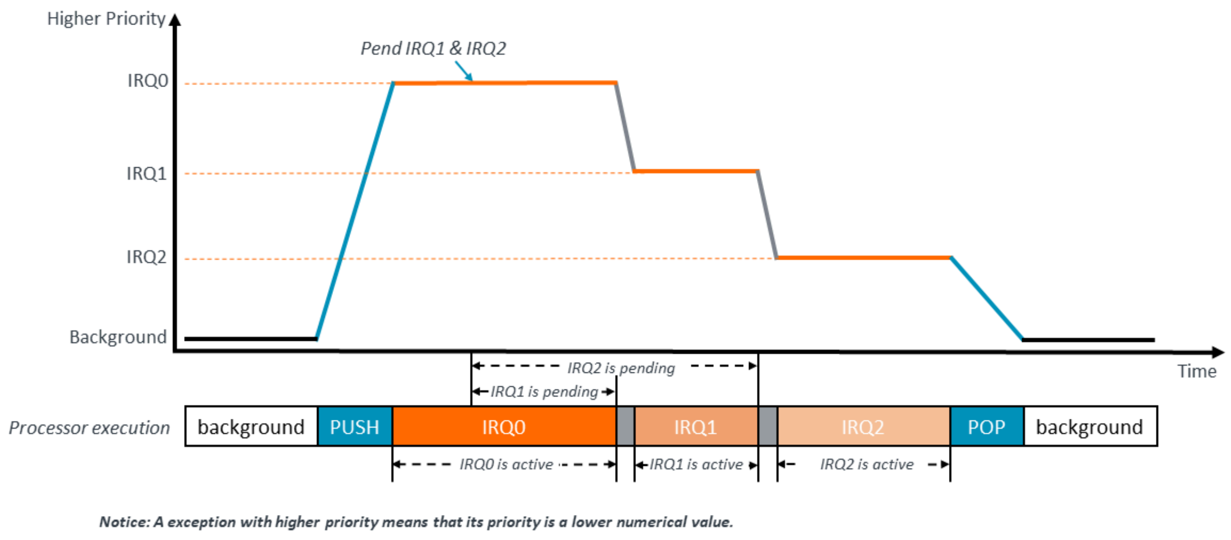
To observe the different behavior, the code in `main()` uses the same methods as case 0 to set the priority levels as `IRQ0 > IRQ1 > IRQ2`. This example triggers IRQ1 and IRQ2 using the IRQ0 handler, as follows:

```
<...>

/* Step2: Set priorities for IRQs */
NVIC_SetPriority(Interrupt0_IRQn, 0x44);
NVIC_SetPriority(Interrupt1_IRQn, 0x66);
NVIC_SetPriority(Interrupt2_IRQn, 0xFF);

/* Pend IRQ1,2 at IRQ0 handler */
IRQPendRequests[InIRQ0Handler] = 1 << (uint32_t)Interrupt1_IRQn|
                                 1 << (uint32_t)Interrupt2_IRQn;

<...>
```

Finally, trigger IRQ0 by setting the pending bit in the NVIC pending register.

The following diagram shows an execution flow chart for this case:

Figure 5-4: Execution flow chart for tail-chaining example



5.2.4.1 Output in target console

The example includes `printf()` functions in the IRQ1 and IRQ2 handlers. For this test, the output console shows the following:

```
We are in IRQ 0 Handler!
Setting IRQ 1 to pend
Setting IRQ 2 to pend
The number of the highest priority pending exception is 17
There is only one active exception.
The number of the highest priority active exception is 16
We are in IRQ 1 Handler!
The number of the highest priority pending exception is 18
There is only one active exception.
The number of the highest priority active exception is 17
We are in IRQ 2 Handler!
There is only one active exception.
The number of the highest priority active exception is 18
Case:2 is completed!
```

Because IRQ0 is the highest priority, it is handled first. In the IRQ0 handler, the active interrupt is 16 (IRQ0) and the highest priority pending exception is 17 (IRQ1).

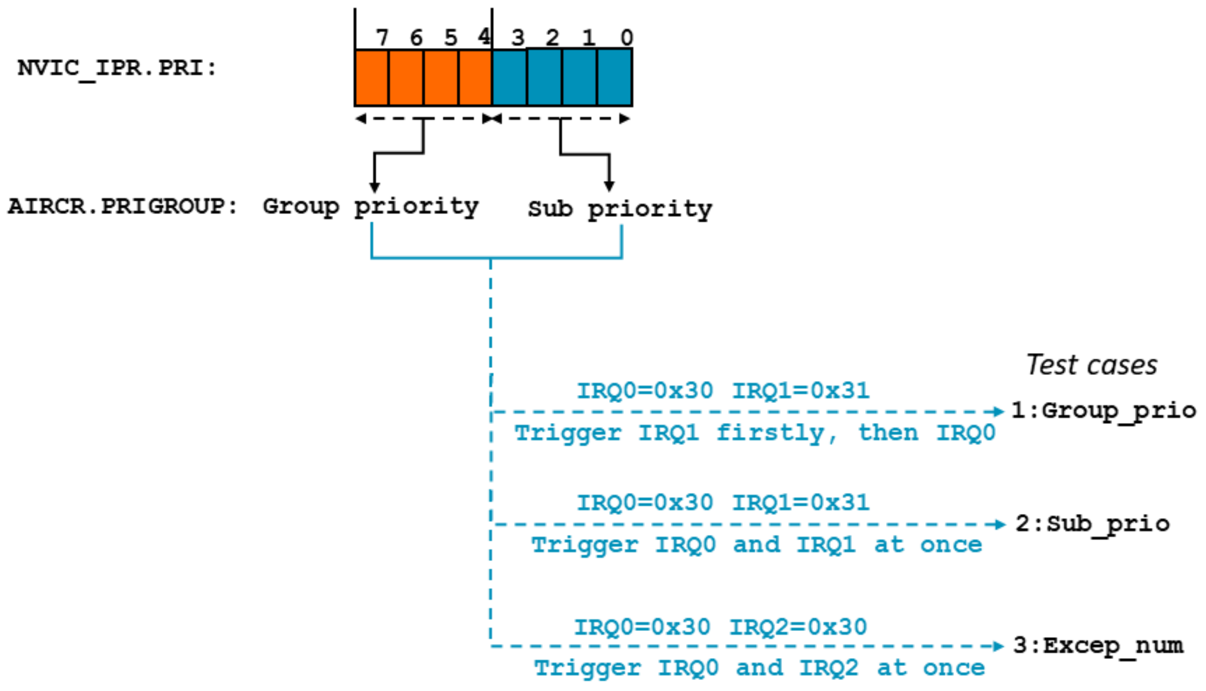
In the IRQ1 handler, the same behavior is observed.

5.2.5 Case 3: effect of group priority and sub-priority

When multiple external interrupts occur, group priority means the pre-emption level. The lower the numerical value, the more important is the interrupt. If two interrupts occur with the same pre-emption level, the interrupt with the lowest sub-priority level is served first. In this case, we test the effect of group priority, sub-priority, and exception number. We can give the interrupts different priority values and change the order in which they are triggered to simulate different tests.

The following flowchart shows how the code for this case executes:

Figure 5-5: Execution flow chart for case3_execution_process



5.2.5.1 Effect of group priority test

This test is a pre-emption scenario where multiple IRQs have the same group priority. Running the test shows the different scenarios where pre-emption happens.

To test the effect of group priority, the code enables two IRQs:

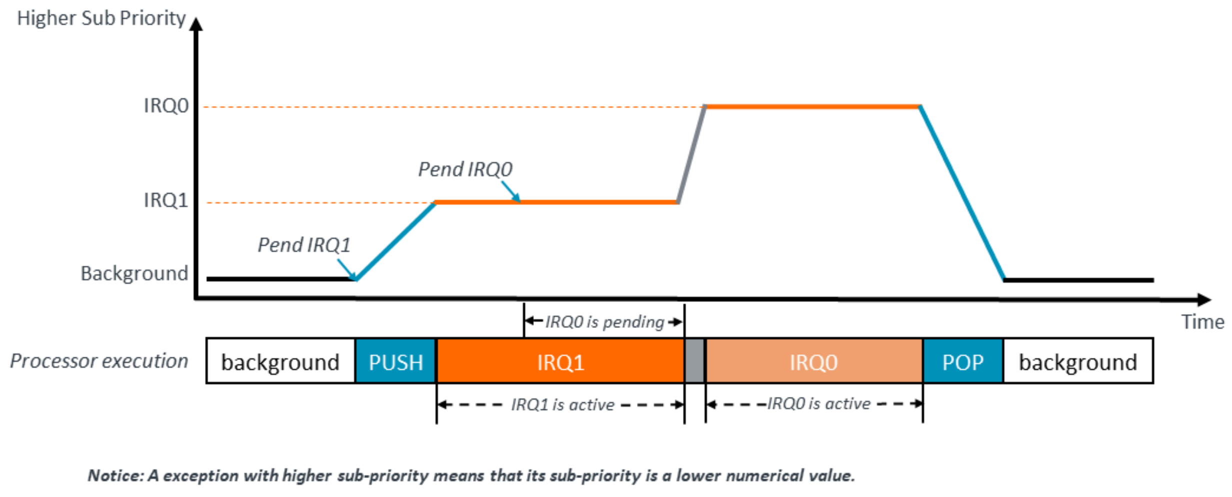
- IRQ0 has group priority 3 and sub-priority 0.
- IRQ1 has group priority 3 and sub priority 1.

The code sets `IRQPendRequest` to trigger IRQ0 using the IRQ1 handler, then triggers IRQ1 to initiate the test:

```
<...>

/* Step2: Set priorities for IRQs */
/* IRQ0: group priority 3, sub priority 0 */
Set_Pri_IRQn(Interrupt0_IRQn, PriGroup, 3, 0);
Set_Pri_IRQn(Interrupt1_IRQn, PriGroup, 3, 1);

IRQPendRequests[InIRQ1Handler] = 1 << (uint32_t)Interrupt0_IRQn;
NVIC->ISPR[0] |= 1 << (uint32_t)Interrupt1_IRQn;
__DSB();
__ISB();
printf("Group priority test is completed! \n");
break;

<...>
```

The following flowchart shows how the code for this case executes:

Figure 5-6: Execution flow chart for case3_group_priority_test



For this test, the output console shows the following:

```
We are in IRQ 1 Handler!
Setting IRQ 0 to pend
The number of the highest priority pending exception is 16
There is only one active exception.
The number of the highest priority active exception is 17
We are in IRQ 0 Handler!
There is only one active exception.
The number of the highest priority active exception is 16
Group priority test is completed!
```

The program handles the two IRQs according to the tail-chaining behavior.

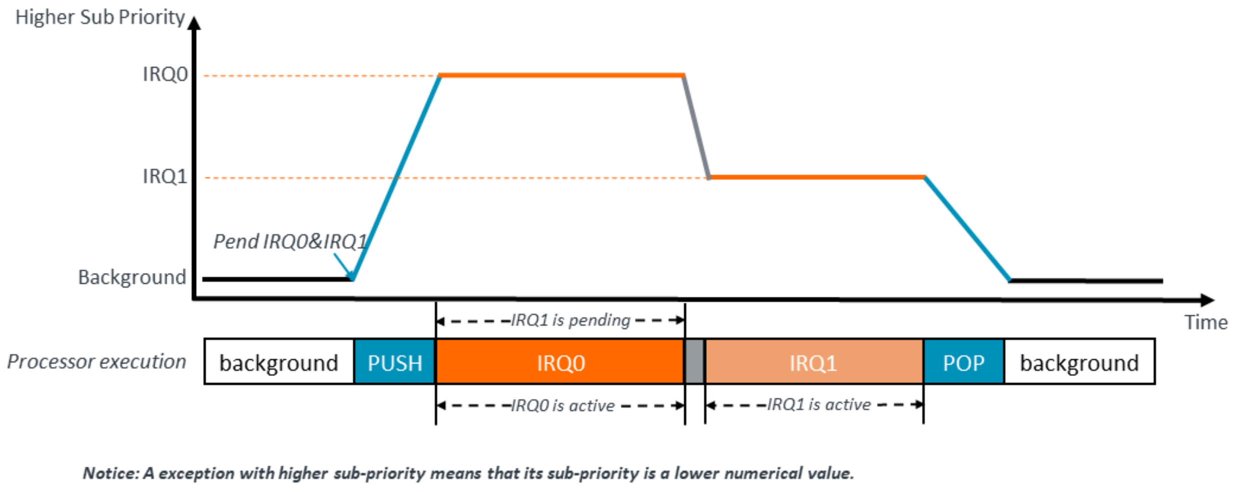
5.2.5.2 Effect of sub-priority test

If multiple pending interrupts have the same group priority, the sub-priority field determines the order in which they are processed. In this example, the code enables two IRQs with the same group priority but different sub-priorities, then triggers them both simultaneously. The numerical value of IRQ0's sub-priority is 0, while IRQ1's sub-priority is 1.

```
<...>

Set_Pri_IRQn(Interrupt0_IRQn, PriGroup, 3, 0);
Set_Pri_IRQn(Interrupt1_IRQn, PriGroup, 3, 1);

NVIC->ISPR[0] |= 1 << (uint32_t)Interrupt0_IRQn|
                 1 << (uint32_t)Interrupt1_IRQn;
__DSB();
__ISB();

<...>
```

The following flowchart shows how the code for this case executes:

Figure 5-7: Execution flow chart for case3_sub_priority_test



For this test, the output console shows the following:

```
We are in IRQ 0 Handler!
The number of the highest priority pending exception is 17
There is only one active exception.
The number of the highest priority active exception is 16
We are in IRQ 1 Handler!
There is only one active exception.
The number of the highest priority active exception is 17
Sub priority test is completed!
```

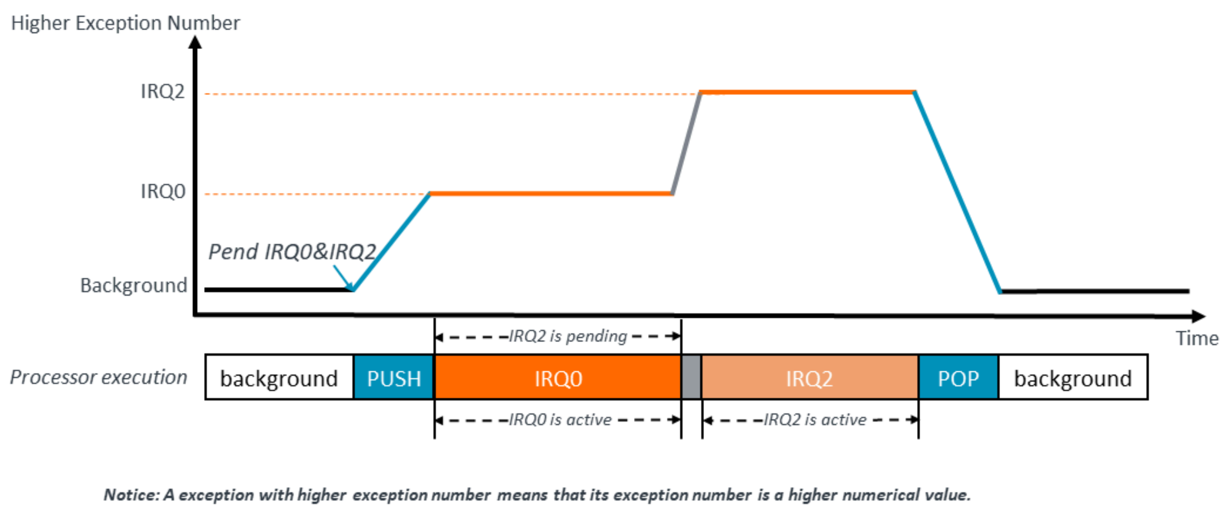
The two IRQs are generated simultaneously, but IRQ0 is handled first because it has the lower sub-priority value.

5.2.5.3 Effect of exception number test

If multiple pending interrupts have the same group priority and sub-priority, the interrupt with the lowest IRQ number is processed first. In this example, the code enables two IRQs with same group priority and sub-priority, then triggers them both simultaneously.

The following flowchart shows how the code for this case executes:

Figure 5-8: Execution flow chart for case3_excep_num_test



For this test, the output console shows the following:

```
We are in IRQ 0 Handler!
The number of the highest priority pending exception is 18
There is only one active exception.
The number of the highest priority active exception is 16
We are in IRQ 2 Handler!
There is only one active exception.
The number of the highest priority active exception is 18
Exception number test is completed!
Case:3 is completed!
Example Project: irq-priority-basic End
```

The two IRQs are generated simultaneously, but IRQ0 is handled first it has the lower exception number.

5.3 priority-boost-types

In some situations, it is useful to be able to prevent higher-priority interrupts from pre-empting a critical section of code. BASEPRI can be used in these situations to set a threshold for interrupts. When privileged code sets BASEPRI, interrupts are masked if they have a priority value greater than or equal to BASEPRI.

This example demonstrates how to set BASEPRI.

The source code for this example is available at Exception_model/priority-boost-types.

5.3.1 Project structure

The file structure of the priority-boost-types example project is as follows:

```
│    IRQSet.h
│    IRQSet.c
│    main.c
└───RTE
    │    RTE_Components.h
    └───Device
        └───ARMv8MML
                |   ARMv8MML_ac6.sct
                |   startup_ARMv8MML.c
                └──system_ARMv8MML.c
```

The files in the example project are as follows:

- `IRQSet.c`: Functions to overwrite the interrupt handler, relocate the vector table, and print the output log.
- `IRQSet.h`: Macro definition and function declarations.
- `main.c`: Similar to the code in the irq-priority-basic example, this code relocates the vector table, configures and triggers a number of IRQs with different priorities, and sets BASEPRI to mask some of these interrupts.
- `RTE/Device/ARMv8MML/startup_ARMv8MML.c`: Configures the vector table, then initializes the MSP and PSP.
- `RTE/Device/ARMv8MML/ARMv8MML_ac6.sct`: Scatter file.
- `RTE/Device/ARMv8MML/system_ARMv8MML.c`: Target definitions.

This is an extension to the irq-priority-basic example, using the same configuration of the MSP and PSP.

5.3.2 IRQ priority and BASEPRI

This example triggers three IRQs to demonstrate how BASEPRI can be used to mask interrupts.

This example uses the same code as described in irq-priority-basic example to relocate the vector table, configure a general external interrupt handler, enable IRQs, and set priorities for them.

The example creates three interrupts with the following priority levels: - IRQ0: `0xFF`, or 224 in decimal. - IRQ1: `0x33`, or 96 in decimal. - IRQ2: `0x22`, or 64 in decimal.

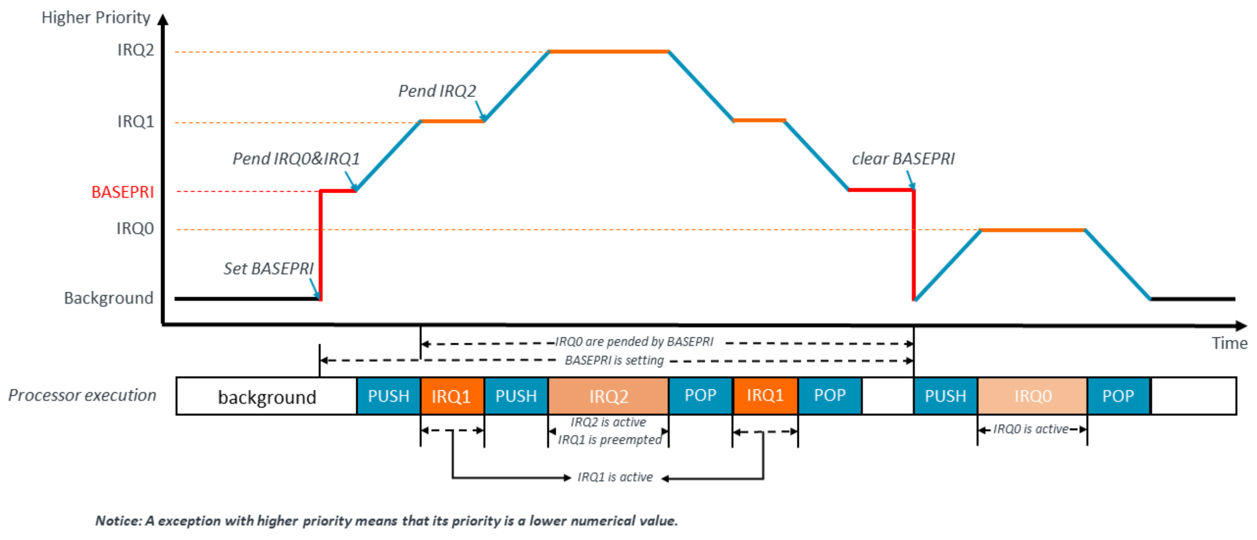
```
<...>
...
NVIC_SetPriority(Interrupt0_IRQn, 0xFF);
NVIC_SetPriority(Interrupt1_IRQn, 0x33);
NVIC_SetPriority(Interrupt2_IRQn, 0x22);
...
<...>
```

Because BASEPRI is set to `0x80`, PE allows only exceptions execution that have a higher priority than this number, such as IRQ1 and IRQ2. These exceptions have a numerical priority value that is lower than BASEPRI. IRQ0 is masked if it is triggered when BASEPRI is set.

The example triggers IRQ0 and IRQ1 first, then pends IRQ2 using the IRQ1 handler. Finally, the code clears BASEPRI to observe the execution of the previously masked IRQ.

The following flowchart shows how the code for this case executes:

Figure 5-9: Execution flow chart for priority-boost-types example



5.3.3 Output in target console

For this example, the output console shows the following:

```
Example Project: priority-boost-types Start
BASEPRI is set with triggering IRQ0 and IRQ1!
We are in IRQ 1 Handler!
Setting IRQ 2 to pend
We are in IRQ 2 Handler!
The number of the highest priority pending exception is 16
There is more than one active exception.
The number of the highest priority active exception is 18
The number of the highest priority pending exception is 16
There is only one active exception.
The number of the highest priority active exception is 17
The number of the highest priority pending exception is 16
BASEPRI is clear!
We are in IRQ 0 Handler!
There is only one active exception.
The number of the highest priority active exception is 16
Example Project: priority-boost-types End
```

First, IRQ0 is masked and pended. Then, IRQ2 pre-empts IRQ1. When the execution is in the IRQ2 handler, there is more than one active exception and the highest priority active exception is IRQ2, with only one pending exception IRQ0. After handling IRQ2, the program handles IRQ1. Then it prints the active and pending status of exception at IRQ1 handler.

After BASEPRI is cleared, the remaining pending exception is handled according to the value of the pre-emption priority. The final output lines show that IRQ0 is active and is going to be handled.

5.4 system-exceptions

This example demonstrates how to generate built-in system exceptions that are commonly used in OS environment, for example PendSV and SysTick.

Privileged software can enable the SysTick interrupt by setting the ENABLE and TICKINT bits in the SysTick Control and Status Register, SYST_CSR.

Privileged software can invoke a PendSV exception by setting ICSR.PENDSEVSET.

These exceptions facilitate task switching in a multi-tasking environment. The SysTick exception provides a periodic interrupt which triggers a thread scheduler that is used to allocate chunks of execution time to a thread, while PendSV performs the actual context switch between threads.

By handing off the actual context switch operation to the PendSV exception and setting that exception to the lowest possible priority, we can guarantee that the state being context switched is always the background thread, because PendSV can not pre-empt any other exceptions.

The source code for this example is available at Exception_model/system-exceptions.

5.4.1 Project structure

The file structure of this system-exceptions example project is as follows:

```
│    main.c
└───RTE
    │    RTE_Components.h
    └───Device
        └───ARMv8MML
              |   ARMv8MML_ac6.sct
              |   startup_ARMv8MML.c
              └───system_ARMv8MML.c
```

The files in the example project are as follows:

- `main.c`: Configures priority settings for SysTick and PendSV, then generates SysTick exceptions using a SysTick counter overflow.
- `RTE/Device/ARMv8MML/startup_ARMv8MML.c`: Configures the vector table, then initializes the MSP and PSP.
- `RTE/Device/ARMv8MML/ARMv8MML_ac6.sct`: Scatter file.
- `RTE/Device/ARMv8MML/system_ARMv8MML.c`: Target definitions.

This example only triggers two exceptions, so we use the default configuration of MSP and PSP.

5.4.2 Triggering system exceptions

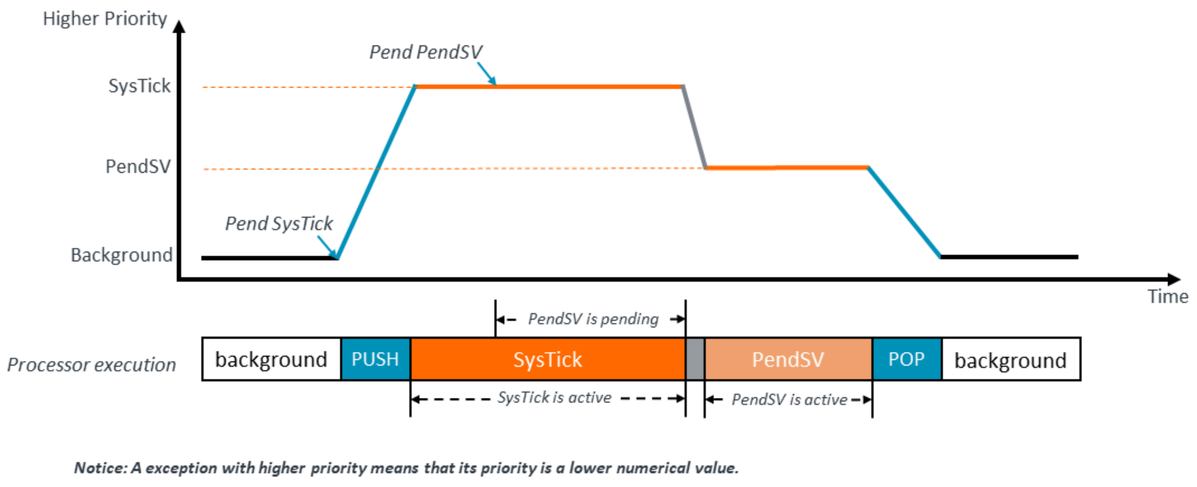
Aside from external interrupts, some of the system exceptions can also have programmable priority level and can have pending status registers.

This example lets us observe the effect of configuring system exception priority. The example sets the priority levels such that `SysTick > PendSV` by calling `NVIC_SetPriority()` function. The highest 3 bits in a byte of System Handler Priority Register (SHPR) set the system exception's priority. This example sets:

- PendSV to have a priority of `0xFF`.
- SysTick has a priority of `0x11`.

The following diagram shows how the code for this example executes:

Figure 5-10: Execution flow chart for system_exceptions example



```
<...>

  /* Step1: Set priority of PendSV and SysTick */
  NVIC_SetPriority(PendSV_IRQn, 0xFF);
  NVIC_SetPriority(SysTick_IRQn, 0x11);

<...>
```

A SysTick exception is generated by a SysTick counter overflow. The code does the following: - Sets the SysTick Reload Value Register is to `0x00FFFFFF`. - Sets the initial SysTick value to zero. - Enables the SysTick IRQ in the SysTick Control and Status Register.

```
<...>

  /* Step2: The RVR is 24bit counter with setting maximum value */
```

```
    SysTick->LOAD = 0x00FFFFFF;

    /* Step3: Generate a SysTick exception after counting 24bit maximum value */
    /* Load the SysTick counter value */
    SysTick->VAL  = 0UL;

    /* Enable SysTick Exception and SysTick Timer */
    SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
                    SysTick_CTRL_TICKINT_Msk |
                    SysTick_CTRL_ENABLE_Msk;

    __DSB();
    __ISB();

  <...>
```

Finally, the code triggers the PendSV by setting the pending bit in the Interrupt Control and State Register (ICSR). To avoid repeated output, the SysTick is closed prior to exception return by disabling `SysTick->CTRL`. The code outputs the pending and active flag for the two handlers.

5.4.3 Output in target console

For this example, the output console shows the following:

```
Example Project: system-exceptions Start
We are in SysTick_Handler!
The pending and active status are
        SCB->ICSR = 0x0000080f
The pending and active status are
        SCB->ICSR = 0x1000e80f
We are in SysTick_Handler end!
We are in PendSV_Handler!
The pending and active status are
        SCB->ICSR = 0x0000080e
Example Project: system-exceptions End
```

The following line of output shows the SysTick exception is active with the exception number 15. Bit 11 shows that there is only one exception active:

```
SCB->ICSR = 0x0000_080f
```

The following line of output shows the SysTick exception is active and there is another exception pending, with the number 14:

```
SCB->ICSR = 0x1000_e80f
```

In PendSV_Handler, the status shows the active exception is number 14 and there is only one exception active.

5.5 svc-number-as-parameter

Cortex-M cores support the SuperVisor Call (SVC) instruction that allows users to trigger an exception. This is useful if the core is in unprivileged mode and the program needs to request a service from the RTOS running in privileged mode. The `svc` instruction has a parameter to select different SVC functions, for example enabling privileged mode. In an operating system environment, the OS provides SVC handler code as a means of providing OS services to user applications.

The svc-number-as-parameter example demonstrates how to select different functions by using a parameter to specify an SVC number.

The source code for this example is available at Exception_model/svc-number-as-parameter.

5.5.1 Project structure

The file structure of the svc-number-as-parameter example project is as follows:

```
│    main.c
│    SelectSVCNumber.h
│    SelectSVCNumber.c
└────RTE
     │    RTE_Components.h
     └────Device
          └────ARMv8MML
                 |    ARMv8MML_ac6.sct
                 |    startup_ARMv8MML.c
                 └────system_ARMv8MML.c
```

The files in the example project are as follows:

- `SelectSVCNumber.c`: Overwrites the SVC handler with an implementation that uses a switching parameter to handle different cases.
- `SelectSVCNumber.h`: Macros and declaration of functions used by `SelectSVCNumber.c`.
- `main.c`: Triggers different SVC exceptions.
- `RTE/Device/ARMv8MML/startup_ARMv8MML.c`: Configures the vector table, then initializes the MSP and PSP.
- `RTE/Device/ARMv8MML/ARMv8MML_ac6.sct`: Scatter file.
- `RTE/Device/ARMv8MML/system_ARMv8MML.c`: Target definitions.

5.5.2 Triggering and handling of SVC exceptions

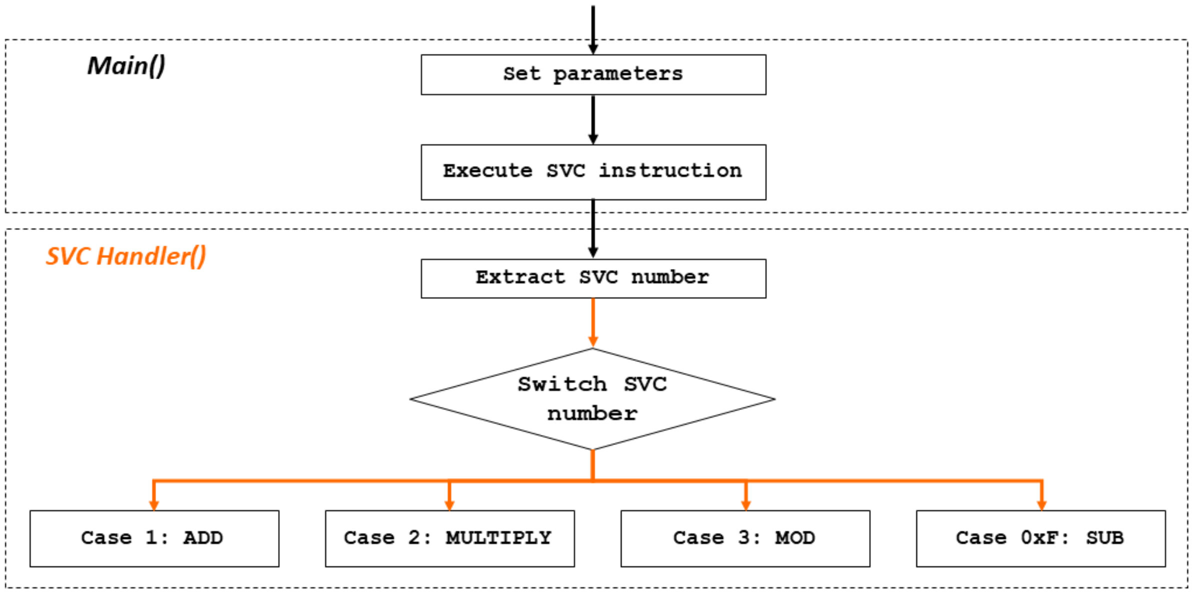
In a high-reliability system, applications run at an unprivileged level while the hardware resources are protected. If an application attempts to access these resources directly, this can result in an access violation which leads to an exception. Applications can call the different services provided by the OS by using the SVC exception handler, using an SVC number to communicate which service is required.

For most exception handlers the entry in the vector table is just a C function. Because we need to pass arguments from the background thread by executing SVC instructions, our SVC handler must be written in assembly. The assembly code passes the stack pointer and the program state used by the background code to the main body of the SVC handler, which is written in C.

The example, when triggering a SVC exception, passes parameters according to the AAPCS compiler rules. The first four register R0-R3 are used to pass argument values into a subroutine. This example passes the address of MSP in R0, and the EXC_RETURN value in R1. The code then uses the address of MSP to retrieve information including the various stacked register values that were automatically saved by hardware. The code then extracts the SVC number using this information, and execute different SVC services depending on the SVC number. This behavior can be seen in the function `SVC_Handler_Main()`. The code uses EXC_RETURN to get the program state from before the `SVC` occurred, for example the SP used and the mode.

The following flowchart shows how the code for this case executes:

Figure 5-11: Execution flow chart for svc-number-as-parameter example



The code for the function `SVC_Handler_Main()` is as follows:

```
<...>

void SVC_Handler_Main(uint32_t *svc_StackFrame, uint32_t EXC_return_value)
{
  /* First argument (r0) is svc_StackFrame
   * Stack contains: r0, r1, r2, r3, r12, LR, PC and xPSR
   * Second argument (r1) is EXC_return_value */
  uint32_t svc_number;
  uint32_t res;
```

```
   uint32_t argValue0 = svc_StackFrame[STK_FRAME_R0];
   uint32_t argValue1 = svc_StackFrame[STK_FRAME_R1];

   ...

   /* Extract lower byte of the SVC opcode to get SVC number */
   svc_number = *((uint16_t *)svc_StackFrame[STK_FRAME_RET_ADDR] - 1) & 0xFF;
   printf("svc_number is %d \n", svc_number);

   /* Switch the number and execute subroutine based on SVC_number */
   switch(svc_number)
   {
      case 1:
        res = argValue0 + argValue1;
        printf("The result of R0+R1 is %d!\n", res);
        break;
      case 2:
        res = argValue0 * argValue1;
        printf("The result of R0*R1 is %d!\n", res);
        break;
      case 3:
        res = argValue0 % argValue1;
        printf("The result of R0 mod R1 is %d!\n", res);
        break;
     ...
   }
}

<...>
```

5.5.3 Output in target console

For this example, the output console shows the following:

```
Example Project: svc-number-as-parameter Start
The return address is 0x00001140
The stacked return state is 0xfffffff9
svc_number is 1
The result of R0+R1 is 12!
The first routine is completed !
The return address is 0x00001156
The stacked return state is 0xfffffff9
svc_number is 2
The result of R0*R1 is 36!
The second routine is completed !
The return address is 0x00001166
The stacked return state is 0xfffffff9
svc_number is 3
The result of R0 mod R1 is 6!
The third routine is completed !
Example Project: svc-number-as-parameter End
```

This example triggers three different SVC exceptions in turn, each specifying a different SVC number as its parameter. In the example handler code, each SVC number selects a different arithmetic operation. For example, executing `svc #1` selects the addition subroutine.

The EXC_RETURN value is `0xffff_fff9`, which means that the SVC exception is taken from the secure world, using the main stack pointer, returning to thread mode, with callee stacked registers.

5.6 synchronous-fault

It is possible to work around UsageFault exceptions by emulating powered off hardware functionality in software. This approach can handle various programming errors including undefined or unsupported instruction opcodes. This means that we can use a synchronous fault handler to deal with these problems, for example by ending the problematic task.

This example triggers a UsageFault by executing a floating-point instruction while the FPU is disabled. The example shows how we can use the exception handler to intercept and fix the fault.

The source code for this example is available from Exception_model/synchronous-fault.

5.6.1 Project structure

The file structure of this example project is here shown:

```
│    main.c
└────RTE
     │    RTE_Components.h
     └────Device
          └────ARMv8MML_DP
                 |   ARMv8MML_ac6.sct
                 |   startup_ARMv8MML.c
                 └───system_ARMv8MML.c
```

- `main.c`: Enable UsageFault, powers off floating-point functionality, and then executes a floating-point instruction to trigger a UsageFault.
- `RTE/Device/ARMv8MML/startup_ARMv8MML.c`: Configures the vector table, then initializes the MSP and PSP.
- `RTE/Device/ARMv8MML/ARMv8MML_ac6.sct`: Scatter file.
- `RTE/Device/ARMv8MML/system_ARMv8MML.c`: Target definitions.

This example only triggers a single UsageFault, so we use the default configuration of MSP and PSP.

5.6.2 Triggering and handling of the UsageFault

This example executes a missing FPU operation to trigger a UsageFault.

The example code does the following:

1. Enables UsageFault by setting the USGFAULTENA bit in the System Handler Control and State Register (SHCSR).
2. Power off floating-point functionality implicitly by setting the SU10 and SU11 bits in the Coprocessor Power Control Register (CPPWR).
3. Execute a floating-point instruction, which triggers an undefined instruction UsageFault.
4. The UsageFault handler clears the SU10 and SU11 bits of the CPPWR to re-enable the FPU.

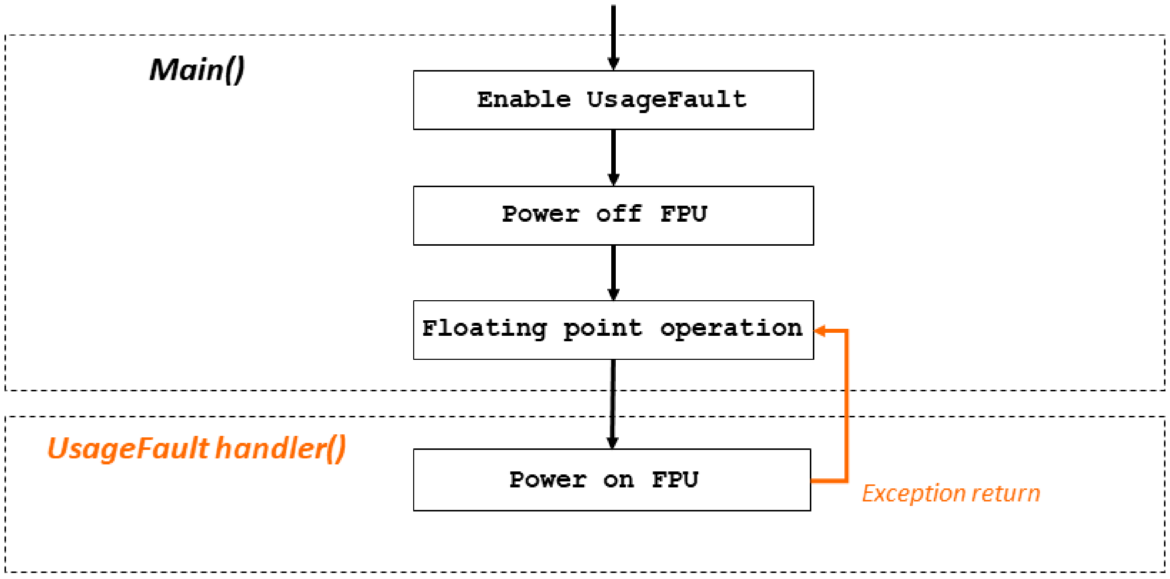
5. The exception returns to the instruction that generated the UsageFault, that is the floating-point instruction, and re-executes this instruction.
6. This time, because the FPU is enabled, the instruction succeeds.

The code that performs the first two steps is as follows:

```
<...>

/* Step1: Enable UsageFault */
SCB->SHCSR |= SCB_SHCSR_USGFAULTENA_Msk ;

/* Step2: Make sure that power off FPU functionality */
SCnSCB->CPPWR |= ((0x1 << 10*2) |
                  (0x1 << 11*2));
__DSB();
__ISB();

<...>
```

The following flowchart shows how the code for this example executes:

Figure 5-12: Execution flow chart for synchronous-fault example



5.6.3 Output in target console

For this example, the output console shows the following when built for the ARMv8MML_DP platform:

```
Example Project: synchronous-fault Start
UsageFault entered!
The UsageFault status register:
```

```
UFSR is 0x8
FPU is enable!
The floating add result is 2.125000
Example Project: synchronous-fault End
```

The fault exception status is obtained by reading the Usage Fault Status Register, part of the Configurable Fault Status Register (CFSR). The output shows that the UFSR value is `0x8` which corresponds to the NOCP bit, indicating that the FPU is disabled. When the FPU is disabled, all floating-point and MVE instructions result in a NOCP UsageFault.

After the exception handler re-enables the FPU by setting the SU10 and SU11 bits, the program re-executes the FPU operation, this time producing the correct result.

5.7 interrupt-deprivileging

This example demonstrates interrupt deprivileging.

In a system, unprivileged code can be placed in independent domains and given limited access to memory and peripherals. To enable services to use interrupts to access peripherals, interrupts must be isolated using the concept of deprivileging interrupts to create a sandbox. For more information about interrupt deprivileging, see What is interrupt deprivileging in Armv8-M?

The source code for this example is available at Exception_model/interrupt-deprivileging.

5.7.1 Project structure

The file structure of the interrupt-deprivileging example project is as follows:

```
|   PrivilegedFuncs.c
|   UnprivilegedFuncs.c
|   excep_prog.h
|   mpu_prog.c
|   mpu_defs.h
|   mpu_prog.h
|   main.c
└───RTE
    |   RTE_Components.h
    └───Device
        └───ARMv81MML_DSP_DP_MVE_FP
              |  ARMv81MML_ac6.sct
              |  startup_ARMv81MML.c
              └──system_ARMv81MML.c
```

- `PrivilegedFuncs.c`: Overwrites the exception handlers.
- `UnprivilegedFuncs.c`: Defines the deprivileging service.
- `excep_prog.h`: Defines the EXC_RETURN macro, configures exception priority, and declares interrupt handling functions.
- `mpu_prog.c`: Sets, switches, and restores MPU regions with specific memory attributes.
- `mpu_defs.h`: Defines memory attributes.

- `mpu_prog.h`: Declares functions used by `mpu_prog.c`.
- `main.c`: Initializes the MPU, sets the priorities for IRQ0, memory management faults, and SVC, then triggers an IRQ to simulate a deprivileging environment.
- `RTE/Device/ARMv81MML/startup_ARMv81MML.c`: Configures the vector table, then initializes the MSP and PSP.
- `RTE/Device/ARMv81MML/ARMv81MML_ac6.sct`: Scatter file.
- `RTE/Device/ARMv81MML/system_ARMv81MML.c`: Target definitions.

5.7.2 Memory region definition

This example defines the following four regions in the scatter file, for dividing the privileged and unprivileged functions:

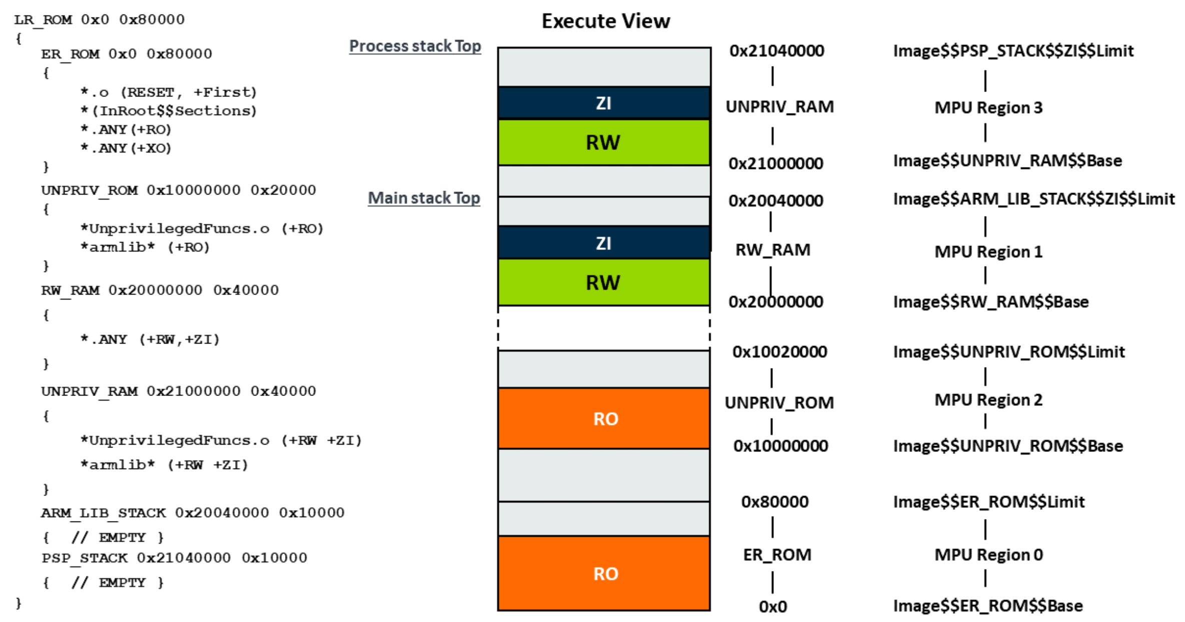
- `LR_ROM`: A read-only but privileged load region. Starts at __RO_BASE (`0x0`). In this example, the load address is same as the execution address, `ER_ROM`.
- `UNPRIV_ROM`: An unprivileged ROM. Unprivileged code is placed in this region.
- `RW_RAM`: Read-write data region. Starts at __RW_BASE. Used for general memory read and write operations.
- `UNPRIV_RAM`: An unprivileged RAM region for the deprivileging service.

After initializing the MPU, the program is in a privileged background code sequence and can only read and execute instructions in privileged mode.

The program needs to enter unprivileged mode to perform the deprivileging service. Therefore, we need another function to change the privilege attribute of both `UNPRIV_ROM` and `UNPRIV_RAM`. This is implemented by `config_MPU()` in `mpu_prog.c`.

The following diagram shows the MPU configuration:

Figure 5-13: MPU region layout for interrupt_deprivileging example



```
LR_ROM 0x0 0x80000
{
   ER_ROM 0x0 0x80000
   {
       *.o (RESET, +First)
       *(InRoot$$Sections)
       *.ANY(+RO)
       *.ANY(+XO)
   }
   UNPRIV_ROM 0x10000000 0x20000
   {
       *UnprivilegedFuncs.o (+RO)
       *armlib* (+RO)
   }
   RW_RAM 0x20000000 0x40000
   {
       *.ANY (+RW,+ZI)
   }
   UNPRIV_RAM 0x21000000 0x40000
   {
       *UnprivilegedFuncs.o (+RW +ZI)
       *armlib* (+RW +ZI)
   }
   ARM_LIB_STACK 0x20040000 0x10000
   {  // EMPTY }
   PSP_STACK 0x21040000 0x10000
   {  // EMPTY }
}
```

5.7.3 Exception configurations

In `main()`, the exception priorities are programmed for IRQ, SVC, and MemManage faults. The exception priority levels are configured such that SVC > MemManage fault > IRQ.

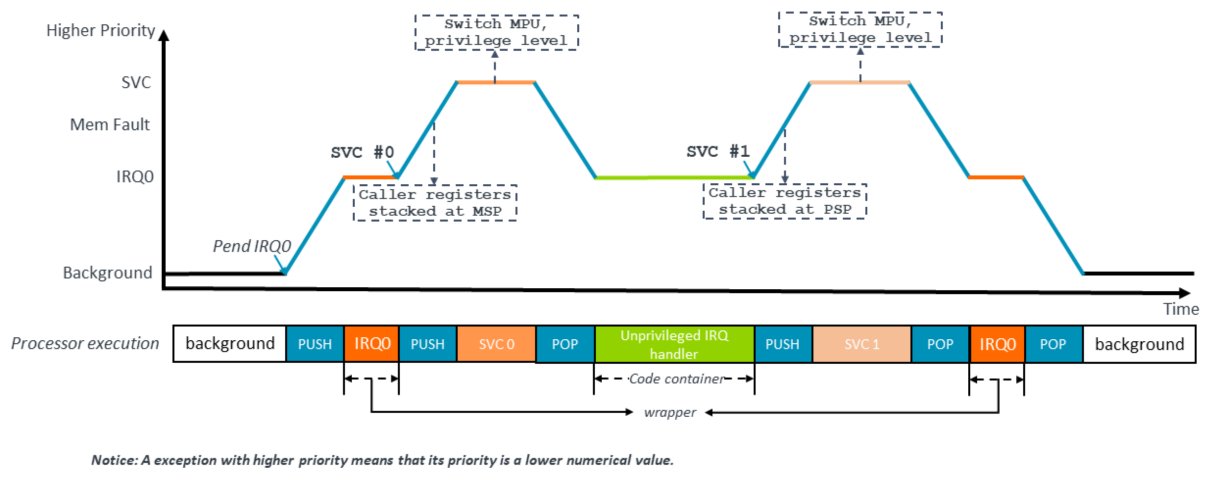
Because only thread mode can be unprivileged, the body of the interrupt handler must run in thread mode. An SVC exception is used to transition from handler mode to thread mode while leaving the original IRQ active. This SVC exception is triggered from a small wrapper that is the initial handler of the IRQ. The MemManage fault handler is set to a high enough priority so that any faults are taken to the MemManage fault handler, and do not escalate to HardFault.

Similarly, to transition back to the wrapper in handler mode and return from the IRQ, another SVC exception is required when the body of the unprivileged IRQ handler has finished.

The code triggers an IRQ as a peripheral interrupt when the program is in privileged mode. After entering the IRQ handler, the code triggers an SVC to perform deprivileging of the exception.

The following diagram shows the priority and execution of each exception.

Figure 5-14: Execution flow chart for the interrupt_deprivileging example


Higher Priority
SVC
Mem Fault
IRQ0
Background
Switch MPU, privilege level
Switch MPU, privilege level
SVC #0
SVC #1
Caller registers stacked at MSP
Caller registers stacked at PSP
Pend IRQ0
Time
Processor execution
background
PUSH
IRQ0
PUSH
SVC 0
POP
Unprivileged IRQ handler
PUSH
SVC 1
POP
IRQ0
POP
background
Code container
wrapper
Notice: A exception with higher priority means that its priority is a lower numerical value.

5.7.3.1 Deprivileging initialization

To prepare for deprivileging, we create a wrapper that is an unprivileged code container with IRQ's priority. First, we trigger IRQ0. The IRQ0 handler needs to save the callee registers R4-R11 and clear the context. This step is to protect the program environment and perform context switching to run the unprivileged context. The code then triggers a SVC exception to request deprivileging. The IRQ0 handler is defined in `PrivilegedFuncs.c` as follows:

```
<...>
__attribute((naked)) void Interrupt0_Handler(void) {
  __asm volatile(
    "PUSH    {R4-R12, LR}                    \n" /* Push the callee regs and keep
 stack pointer 8 byte alignment. */
    "CLRM    {R1-R12}                        \n" /* Clear all the regs */
    "SVC     #0                              \n" /* Request a depriv of the
 execution */
    "POP     {R4-R12, LR}                    \n"
    "BX      LR                              \n"
  );
}
<...>
```

5.7.3.2 Triggering SVC #0 as a deprivileging process

As we saw in the `svc_number_as_parameter` example, the SVC handler in `excep_prog.c` checks the current stack, discovers whether to enable the default register stack, then branches to `SVCHandlerMain` to switch the SVC number and execute the subroutine. The `0` option indicates that we are performing interrupt deprivileging, therefore we need to prepare the deprivileged environment. The code also passes the value of the current stack pointer, MSP, and the current value of EXC_RETURN.

The code performs the following steps:

1. Record information about the PSP which is used by the unprivileged code and also when re-privileging in the second SVC handler to restore state before the SVC exception occurs.

```
<...>
/* Re-use the SVC caller register stack */
/* R1 stores current psp, R2 stores psplim */
/* Refresh the LR, return address and xPSR */
svc_StackFrame[STK_FRAME_R1] = (uint32_t) psp;
svc_StackFrame[STK_FRAME_R2] = (uint32_t) pspLim;
svc_StackFrame[STK_FRAME_LR] = EXCReturn;
<...>
```

1. Allocate the exception return stack frame on PSP that is used when returning from the `svc` to thread mode. The values in this stack frame become the initial state used for that thread.

```
<...>
deprivThreadStackPtr        -= CalleeRegNum;
psp                          = deprivThreadStackPtr;
<...>
```

1. Update the execution for unprivileged thread mode by setting the return address as the unprivileged function address and xPSR as the program status.

```
<...>
psp[STK_FRAME_RET_ADDR]      = (uint32_t)&depriv_service;
psp[STK_FRAME_XPSR]          = svc_StackFrame[STK_FRAME_XPSR] & 0xFFFFFE00;
<...>
```

1. Reprogram the MPU for the new unprivileged thread by changing memory attributes, the value of process stack, and privilege level.

```
<...>
/* Preparation for fake exception return */
/* Switch to the unprivileged memory region */
config_MPU(ARM_MPU_NON_PRIV);
__set_PSP((uint32_t)psp);
__set_PSPLIM((uint32_t)deprivThreadStack);
__set_CONTROL(__get_CONTROL() | CONTROL_nPRIV_Msk);
<...>
```

EXC_RETURN (`excReturn`) is passed using the AAPCS rules and read from the R0 register. When the `BX R0` instruction at the end of `SVC_Handler(void)` is executed, the exception returns. The program returns to unprivileged thread mode with the same priority of IRQ0.

We can add other operations to the deprivileged service routine, for example communicating with other peripherals or printing the log.

5.7.3.3 Re-privileging process

After executing the code in unprivileged mode, the code calls `depriv_return()` to run an SVC instruction (`__asm("SVC #1")`) and return to IRQ handler mode.

```
<...>

__attribute__((naked)) void depriv_return(){
  __asm volatile(
    "SVC     #1       \n"
  );
}

void depriv_service(void){
...
  depriv_return();
}

<...>
```

The SVC handler that performs the re-privileging must restore the state that was saved during the first SVC invocation, that is the SVC handler that performed the deprivileging. This includes restoring the previous PSP, PSPLim, and EXC_RETURN values.

Next, the code restores the MPU configuration, restore the PSP, and switches to privileged mode. Executing `BX R0` returns the program to the original IRQ0 handler.

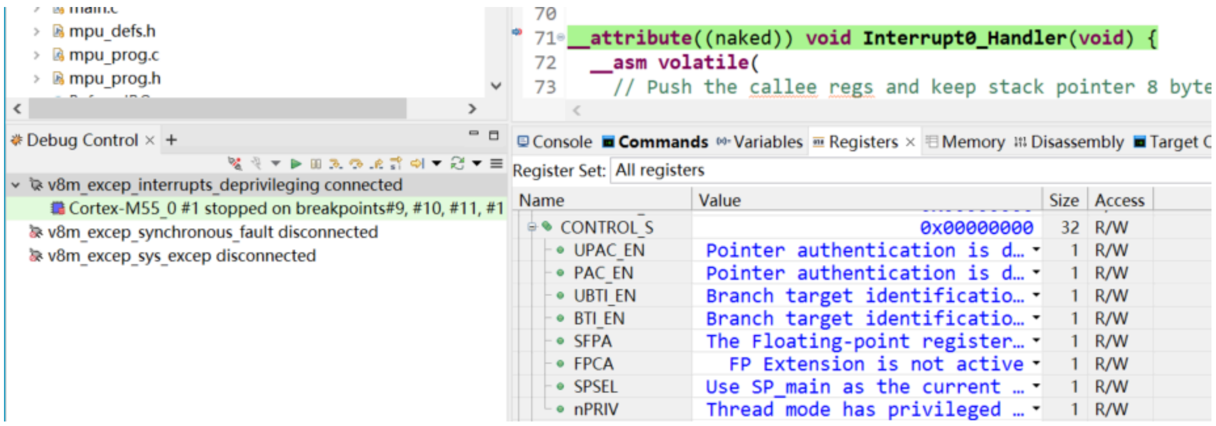
```
<...>
case 1: /* IRQ reprivileging request */
    /* From thread mode to handler mode, return to IRQ
    * Re-use the SVC #0 caller register stack to complete a fake exception return */
    excReturn                  = msp[STK_FRAME_LR];
    psp                        = (uint32_t *) msp[STK_FRAME_R1];
    pspLim                     = (uint32_t *) msp[STK_FRAME_R2];
    deprivThreadStackPtr       = svc_StackFrame + CalleeRegNum;

    /* Preparation for fake exception return */
    config_MPU(ARM_MPU_PRIV);
    __set_PSP((uint32_t)psp);
    __set_PSPLIM((uint32_t)pspLim);
    __set_CONTROL(__get_CONTROL() & ~CONTROL_nPRIV_Msk);
    break;
<...>
```

5.7.4 Output in target console

This example uses two breakpoints:

- The first breakpoint is set before entering the IRQ exception. The program stops in handler mode, when the control register contains `0x0`, as seen in the following screenshot:

Figure 5-15: IRQ exception breakpoint



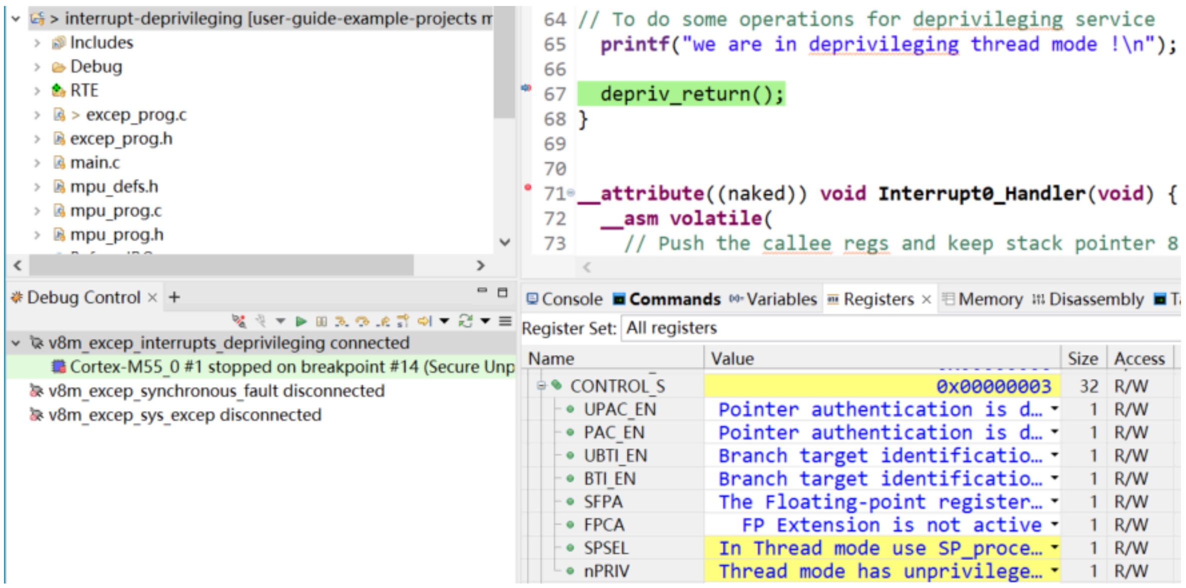
```
__attribute((naked)) void Interrupt0_Handler(void) {
  __asm volatile(
   // Push the callee regs and keep stack pointer 8 byte
```

v8m_excep_interrupts_deprivileging connected
Cortex-M55_0 #1 stopped on breakpoints#9, #10, #11, #1
v8m_excep_synchronous_fault disconnected
v8m_excep_sys_excep disconnected

Name	Value	Size	Access
CONTROL_S	0x00000000	32	R/W
UPAC_EN	Pointer authentication is d…	1	R/W
PAC_EN	Pointer authentication is d…	1	R/W
UBTI_EN	Branch target identificatio…	1	R/W
BTI_EN	Branch target identificatio…	1	R/W
SFPA	The Floating-point register…	1	R/W
FPCA	FP Extension is not active	1	R/W
SPSEL	Use SP_main as the current …	1	R/W
nPRIV	Thread mode has privileged …	1	R/W

- The second breakpoint is set before entering the SVC1 exception. The program finishes the deprivileging service and prints the state at completion. On returning from the SVC0 handler, the program is in the sandbox. The mode and SP have changed, and the program stops in unprivileged thread mode. At this point, the control register is `0x3`, as seen in the following screenshot:

Figure 5-16: SVC1 exception breakpoint



```
// To do some operations for deprivileging service
  printf("we are in deprivileging thread mode !\n");

  depriv_return();
}


__attribute((naked)) void Interrupt0_Handler(void) {
  __asm volatile(
   // Push the callee regs and keep stack pointer 8
```

v8m_excep_interrupts_deprivileging connected
Cortex-M55_0 #1 stopped on breakpoint #14 (Secure Unp
v8m_excep_synchronous_fault disconnected
v8m_excep_sys_excep disconnected

Name	Value	Size	Access
CONTROL_S	0x00000003	32	R/W
UPAC_EN	Pointer authentication is d…	1	R/W
PAC_EN	Pointer authentication is d…	1	R/W
UBTI_EN	Branch target identificatio…	1	R/W
BTI_EN	Branch target identificatio…	1	R/W
SFPA	The Floating-point register…	1	R/W
FPCA	FP Extension is not active	1	R/W
SPSEL	In Thread mode use SP_proce…	1	R/W
nPRIV	Thread mode has unprivilege…	1	R/W

The target console displays the status:

```
we are in deprivileging thread mode !
```

After all exceptions have completed, the state is restored to `0x0`. The target console shows the following output:

```
Example Project: interrupt-deprivileging Start
we are in deprivileging thread mode !
we are back from IRQ!
The status is 0xc
Example Project: interrupt-deprivileging End
```

Note

For Armv7-M, the CCR.NONBASETHRDENA bit controls whether the processor can enter Thread mode at an execution priority level other than base level. To run this example using a device based on Armv7-M, this bit must be enabled. For more information, see Exception return behavior in the ARMv7-M Architecture Reference Manual.

5.8 context-switch-fp

In a typical operating system, round-robin scheduling can be used to arrange tasks equally using time slices. The time slices are defined using a special timer, SysTick, which produces a periodic interrupt. When a SysTick exception occurs, a PendSV with a lower priority is triggered to switch contexts and move to the next task.

When switching contexts, the contents of registers must be saved. Typically, R4-R11 are saved at context switching. However, when an FPU is enabled, an extra 34 registers need to be stacked when switching tasks.

However, because FPU registers are infrequently used by ISRs, there is an optimization that can be employed to reduce the overhead of context switching. This optimization is called lazy floating-point state preservation. This optimization defers stacking of the FPU registers until a floating-point instruction is used in the exception. If no floating-point instructions are used in the ISR, as is often the case, then the FPU registers never need to be stacked. Removing the need to push and pop these registers reduces interrupt latency, which is especially important for an RTOS.

This example shows how to use the SysTick exception to switch tasks, and demonstrates the lazy floating-point state preservation optimization. It is a simplification of what a real RTOS does, to illustrate how context switching is performed.

The source code for this example is available at Exception_model/context-switch-fp.

5.8.1 Project structure

The file structure of the context-switch-fp example project is as follows:

```
|   scheduler.c
|   scheduler.h
|   task1.c
```

```
│   task2.c
│   main.c
└───RTE
    │   RTE_Components.h
    └───Device
        └───ARMv8MML_DP
                |   ARMv8MML_ac6.sct
                |   startup_ARMv8MML.c
                └───system_ARMv8MML.c
```

The files in the example project are as follows:

- `scheduler.c`: Overwrites the exception handlers to implement task switching and initialize the stack.
- `scheduler.h`: Declares functions used by `scheduler.c`.
- `task1.c`: Defines the task1 function, which calculates the diagonal of a square.
- `task2.c`: Defines the task2 function, which calculates the area of a circle.
- `main.c`: Sets the priorities for both PendSV and SysTick, initializes the tasks, and starts the scheduler.
- `RTE/Device/ARMv8MML/startup_ARMv8MML.c`: Configures the vector table, then initializes the MSP and PSP.
- `RTE/Device/ARMv8MML/ARMv8MML_ac6.sct`: Scatter file.
- `RTE/Device/ARMv8MML/system_ARMv8MML.c`: Target definitions.

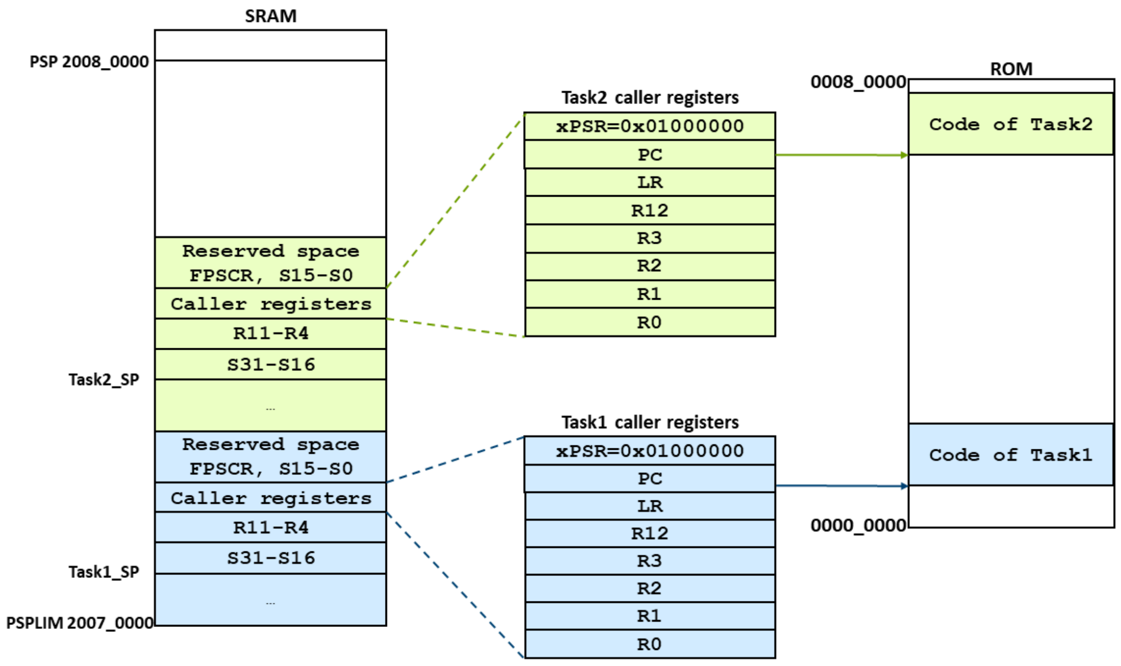
5.8.2 Task initialization

Initializing a task reserves memory for that task's stack. The code reserves 2KB for each new task on the PSP stack and saves the pointer address in the task's data structure. Task number TaskID has a starting stack pointer address calculated as follows:

```
Tasks[TaskID].sp        = PSPBase + TaskID * TASK_STACK_SIZE - NORM_STACK_FRAME_SIZE;
```

The following diagram shows how the stack is used in this example:

Figure 5-17: Stack usage in the context-switch-fp example



When switching to a task, the processor uses the PSP pointer to find the following information:

- Last status, loaded to the xPSR register.
- Next instruction to be executed, loaded to the PC register.
- Return mode, loaded to the EXC_RETURN value.
- Last execution status, loaded to the R0 to R12 registers.

Because the example uses the PendSV exception to trigger context switching, the following registers are automatically saved at exception entry, in fixed order:

1. xPSR
2. PC
3. LR
4. R12
5. R3
6. R2
7. R1
8. R0

To start the first task from the main thread, the code initializes the following registers:

- Set xPSR to `0x01000000`.

No special status should be set at the initial stage of a task, apart from setting the Thumb state bit.

- Set PC to the address of the task's handler function.
- Set LR to the address of the TerminateTask function.

  If the task thread returns, the thread terminates.

The other registers, such as R4-R11, use their default values.

This portion of the code is as follows:

```
    <...>
    /* Refresh value for exception return */
    #define XPSR_THREAD                  0x01000000
    #define EXC_RETURN_THREAD_S_PSP      0xFFFFFFED
    ...

    void InitTask(void* task, uint32_t TaskID){
    ...
      Tasks[TaskID].sp        = PSPBase + TaskID * TASK_STACK_SIZE -
  NORM_STACK_FRAME_SIZE;
      Tasks[TaskID].spLimit   = PSPBase + (TaskID - 1) * TASK_STACK_SIZE;
      Tasks[TaskID].excReturn = EXC_RETURN_THREAD_S_PSP;

      /* Initialize thread stack */
      Tasks[TaskID].sp[STK_FRAME_XPSR]      = XPSR_THREAD;
      Tasks[TaskID].sp[STK_FRAME_RET_ADDR] = (uint32_t)task;
      Tasks[TaskID].sp[STK_FRAME_LR]        = (uint32_t)TerminateTask;
    }
    <...>
```

5.8.3 Start scheduling

The example configures SysTick to generate a periodic exception, using `SystemCoreClock` like a timer to finish task switching.

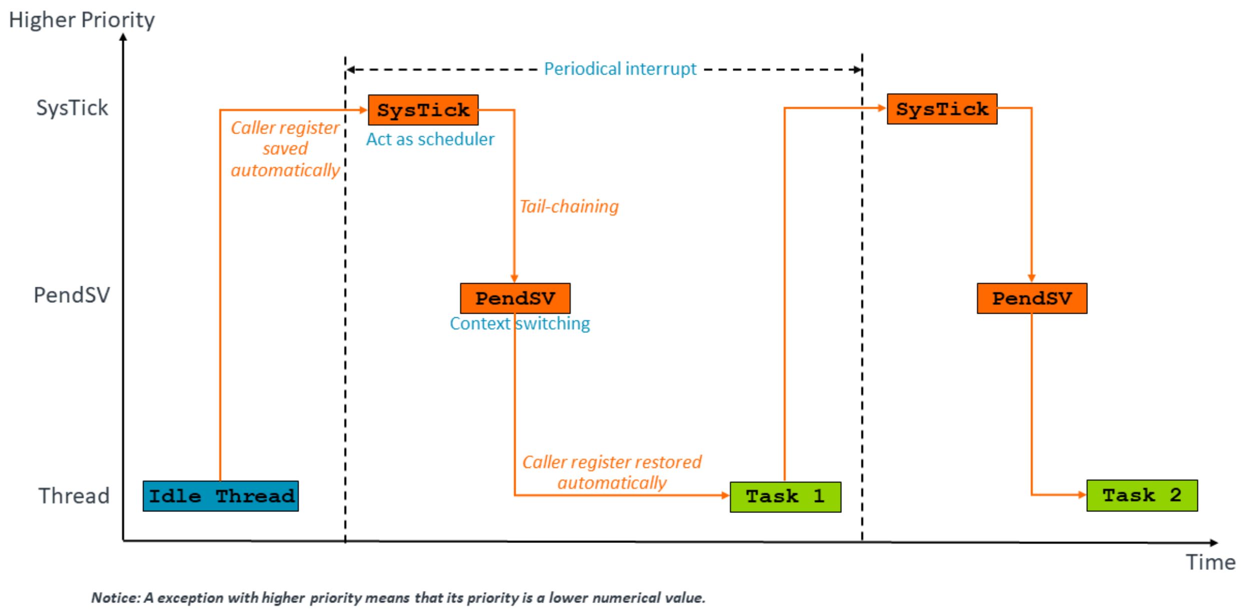
Before the first exception occurs, the program is in an idle thread. Execution remains in a while loop to simulate the idle thread, until the first exception occurs and the program schedules the first task.

5.8.4 Exception configurations

In `main()`, PendSV is set to the lowest priority level, so that it can not pre-empt any other exceptions. SysTick is set to a higher priority level. At the end of `main()`, the scheduler starts, using SysTick as the timer for context switching by calling the `SysTick_Config()`. When a SysTick exception occurs, the pending bit of PendSV is set. The program enters the PendSV handler to complete the context switch.

The following diagram shows the priority and execution flow of each exception:

Figure 5-18: Execution flow chart for context-switch-fp example

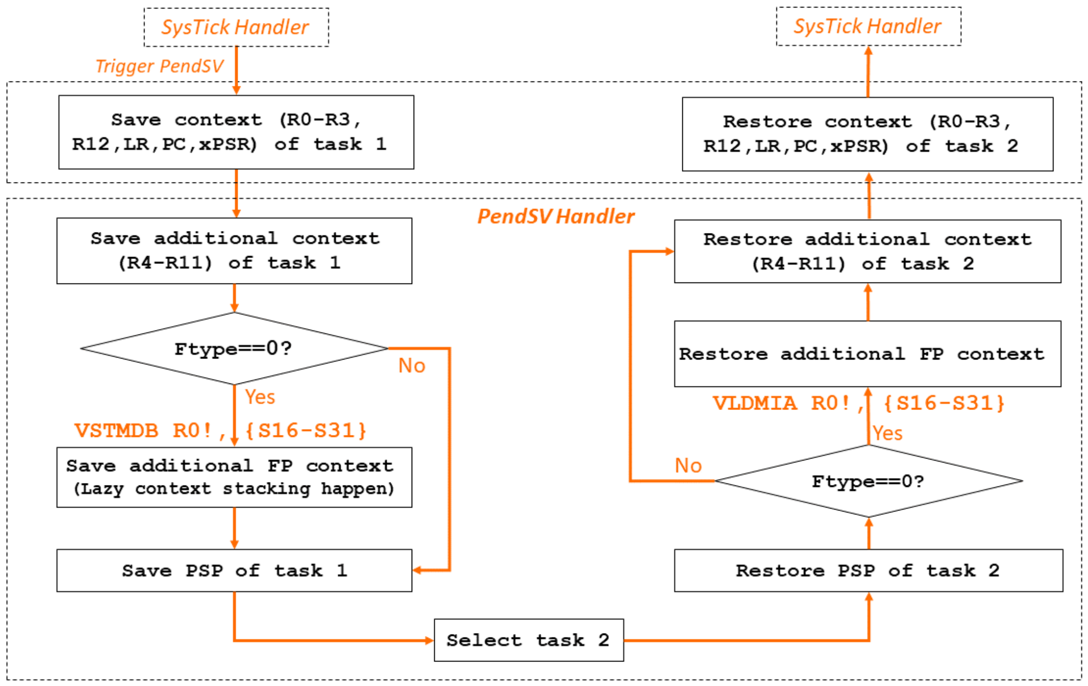

Higher Priority
SysTick
PendSV
Thread
Time
Periodical interrupt
Caller register saved automatically
SysTick
Act as scheduler
Tail-chaining
PendSV
Context switching
Caller register restored automatically
Idle Thread
Task 1
SysTick
PendSV
Task 2
Notice: A exception with higher priority means that its priority is a lower numerical value.

The PendSV handler retrieves stacked information including the execution state of the task and the values of the active hardware registers. The handler does the following:

1. Read EXC_RETURN.SPSEL to determine whether to use the main stack or the process stack to record context such as EXC_RETURN and the return address. If SPSEL is 0, tasks are executed from the main thread. Otherwise the program is going from one task to another task.
2. The caller registers are automatically saved and restored on exception entry and exit. However, the callee registers such as R11-R4 need to be stacked manually. The processor uses a full descending stack, therefore the code uses `STMDB`, Store memory using Decrement Before, to save registers on the stack. Note that the write back option of the STMDB instruction is used, signified by the ! at the end of the register list, to update the value of the stack pointer.
3. If the thread uses the FPU then the lazy stacking feature preserves registers S0-S15 when the handler attempts to access the FP registers. The software must manually stack the remaining registers, S16-S31. It is the act of the software manually saving these higher registers that triggers the lazy state preservation of the lower registers.
4. After saving the context of the first task, the program restores the context of the next task. The PendSV handler decides which task to run next, then gets the task ID and its context thread. The code then restores the context and running state from the thread stack, and the new task executes.

The following diagram shows the context switching process:

Figure 5-19: Execution flow chart for the context-switch-fp example


SysTick Handler
Trigger PendSV
Save context (R0-R3, R12,LR,PC,xPSR) of task 1
Save additional context (R4-R11) of task 1
Ftype==0?
No
Yes
VSTMDB R0!, {S16-S31}
Save additional FP context (Lazy context stacking happen)
Save PSP of task 1
Select task 2
PendSV Handler
SysTick Handler
Restore context (R0-R3, R12,LR,PC,xPSR) of task 2
Restore additional context (R4-R11) of task 2
Restore additional FP context
VLDMIA R0!, {S16-S31}
Yes
No
Ftype==0?
Restore PSP of task 2

5.8.5 Output in target console

For this example, the output console shows the following:

```
Example Project: context-switch-fp Start
Start Scheduler !

we are in SysTick handler !

The diagonal of a square with side=1: 0.000000

The diagonal of a square with side=2: 1.414214

The diagonal of a square with side=3: 2.828427

The diagonal of a square with side=4: 4.242641
we are in SysTick handler !

The area of a circle with r=2: 3.141593

The area of a circle with r=3: 12.566371

The area of a circle with r=4: 28.274334

The area of a circle with r=5: 50.265484

...
```

The output first shows the program switching to task1 from the main thread. Then the two tasks alternate in round-robin style.

Note

There is no guarantee that exceptions occur neatly between messages in the output. It is likely you will see interrupted messages when exceptions occur while a message is being output, because the different tasks both use the same output log.

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