ARMv8-M Exception handling

Version 2.0

Revision Information

The following revisions have been made to this User Guide.

Date	Issue	Confidentiality	Change
02 September 2016	0100	Non-Confidential	First release
28 February 2017	0200	Non-Confidential	Second release

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM® in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

Confidentiality Status

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

Unrestricted Access is an ARM internal classification.

Product Status

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

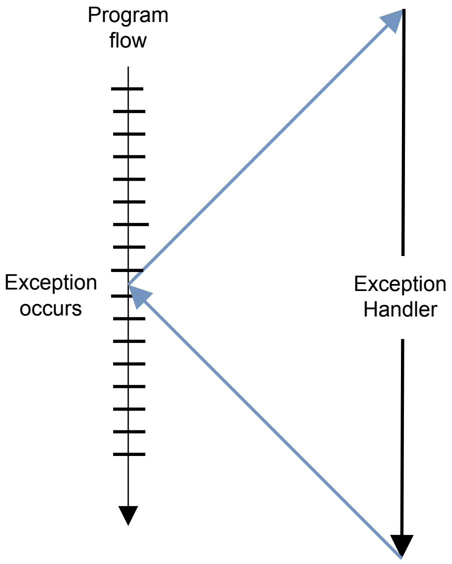
Web Address

http://www.arm.com

Contents

1 ARMv8-M Exception handling

An exception is any condition that requires the core to stop normal execution and instead execute a dedicated software routine that is known as an exception handler.



Exceptions are conditions or system events that usually require remedial action or an update of system status by privileged software to ensure smooth functioning of the system. This is called handling an exception.

The ARMv8-M exception model describes how the processor responds to an exception, the properties that are associated with each exception, such as its priority level, and the exception return behavior. There is typically an exception handler that is associated with each exception type.

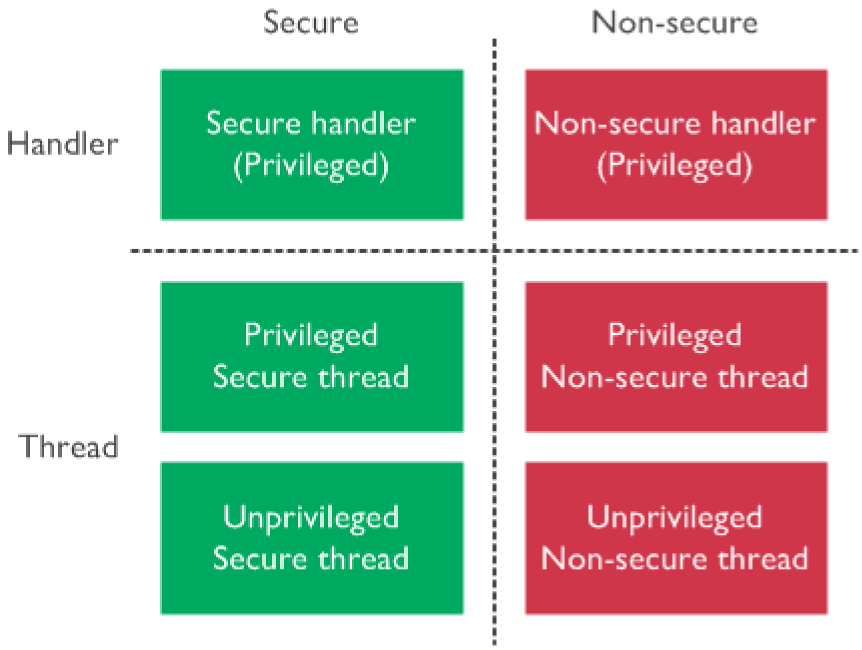
There are differences between ARMv8-M processors and other ARM processor families; for example, there are no IRQ or FIQ handling modes. ARMv8-M processors also have an integrated nested Vectored Interrupt Controller (NVIC), which identifies external interrupts to the core. This is unlike other ARM processors which have little direct hardware support for interrupts.

Hardware external to the processor core usually signals an interrupt, but there are several other events that can cause the core to take an exception, for example, internal events such as OS calls using the `SVC` instruction.

If the ARMv8-M Security Extension is implemented, it modifies some aspects of exception handling.

- Some exception handling resources can be either Secure or Non-secure.
- Some resources are banked (duplicated) to have both a Secure and Non-secure version.
- Additional memory mapped registers are introduced in the System Control Space.
- Additional bits are populated in some control and status registers.

The processor mode, states, and privilege level provide multiple possible combinations:


Secure
Non-secure
Handler
Secure handler (Privileged)
Non-secure handler (Privileged)
Thread
Privileged Secure thread
Privileged Non-secure thread
Unprivileged Secure thread
Unprivileged Non-secure thread

The processor mode changes when an exception occurs in Thread mode, but does not change when an exception occurs in Handler mode.

To separate Secure and Non-secure stacks, processors that are based on the ARMv8-M architecture support four stack pointers) if the ARMv8-M architecture with Security Extension is implemented:

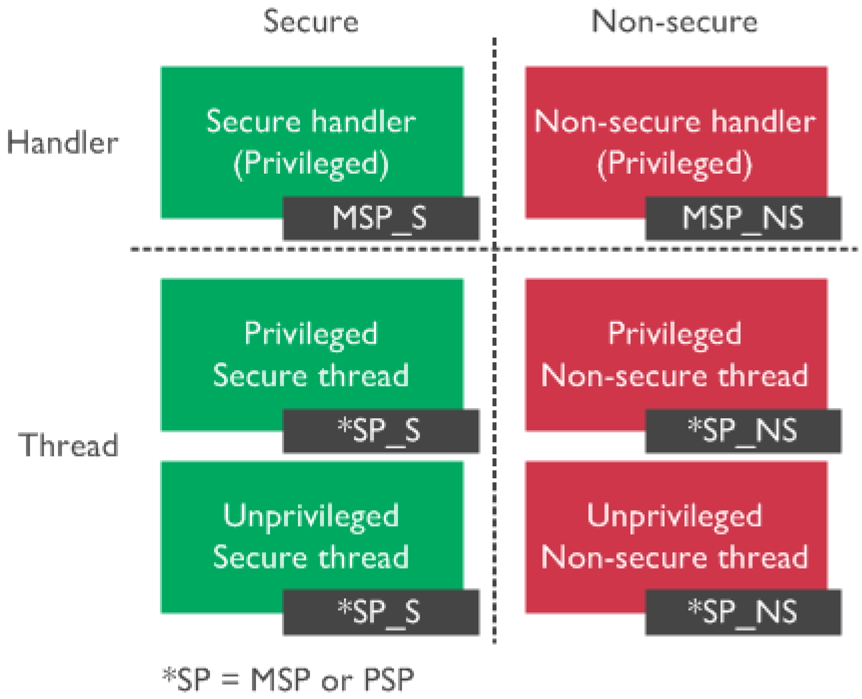
- Main Stack Pointers(MSP).
- Process Stack Pointers(PSP).

In addition, a stack limit feature is provided using stack limit registers (accessible using MSR and MRS instructions) in Privileged level.

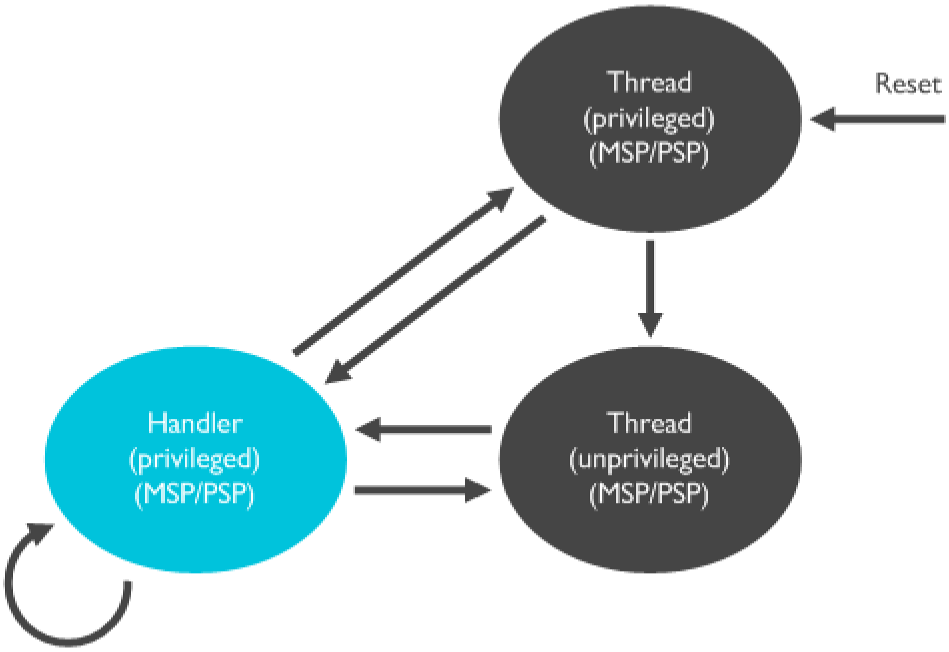
The following table summarizes the ARMv8-M architecture support for stack pointers.

Stack	Stack pointers	Corresponding stack limit register
Secure Main Stack – used by Secure handlers, and Secure thread.	MSP_S	MSPLIM_S
Secure Process Stack – used by Secure threads.	PSP_S	PSPLIM_S
Non-secure Main Stack – used by Non-secure handlers, and Non-secure thread.	MSP_NS	MSPLIM_NS ARMv8-M with Main extension only
Non-secure Process Stack – used by Non-secure threads.	PSP_NS	PSPLIM_NS ARMv8-M with Main extension only

The _S and _NS suffixes are used in the ARMv8-M architecture to identify whether the resource is for the Secure state or Non-secure state. Stack limit registers are available for all stack pointers in the ARMv8-M architecture with Main Extension. In the ARMv8-M architecture, the stack limit registers are available for Secure stack pointers only.


Secure
Non-secure
Handler
Secure handler (Privileged)
MSP_S
Non-secure handler (Privileged)
MSP_NS
Thread
Privileged Secure thread
*SP_S
Privileged Non-secure thread
*SP_NS
Unprivileged Secure thread
*SP_S
Unprivileged Non-secure thread
*SP_NS
*SP = MSP or PSP

The following sequence and figure show the possible stages in exception handling.


Thread (privileged) (MSP/PSP)
Reset
Handler (privileged) (MSP/PSP)
Thread (unprivileged) (MSP/PSP)

From Reset:

At reset all interrupts are disabled.

- The core enters Privileged Thread mode (MSP).

- An internal exception is taken:
    - Switch to Handler mode.
    - Run an exception handler.
    - Return to Privileged Thread Mode (Privileged +Thread+MSP).
- Change from PT(Privileged +Thread) to UT(Unprivileged + Thread).

For an external exception.

- Switch to Handler mode.
- Run exception handler.
- Exception handler is interrupted.
- Reenter Handler mode
- Service higher priority interrupt
- Return to Handler mode.
- Return to UT.

2 Types of exception

The exception model for the ARMv8‑M architecture provides various types of exception. Each exception is handled in turn before returning to the original application. When multiple exceptions occur simultaneously, they are handled in a fixed order of priority.

The following types of exception exist:

Reset

Reset is invoked on power-up or a Warm reset. The ARMv8‑M exception model treats reset as a special form of exception. It is permanently enabled. When reset occurs, the operation of the processor stops. This could be at any point in instruction execution. When the processor restarts, execution restarts from the address that is provided by the reset entry in the vector table. Execution restarts as privileged execution in Thread mode.

NMI

A Non-maskable Interrupt (NMI) is signaled by a peripheral or triggered by software. This is the highest priority exception other than reset. It is permanently enabled. NMIs cannot be:

- Masked or prevented from activation by any other exception.
- Preempted by any exception other than Reset.

HardFault

A HardFault is an exception that occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism.

In the ARMv8‑M architecture a Security violation in a Non-secure NMI handler would trigger and be preempted by a Secure HardFault exception.

MemManage

A MemManage fault is an exception that occurs because of a memory protection-related fault. The MPU or the fixed memory protection constraints determines this fault, for both instruction and data memory transactions. This fault is always used to abort instruction accesses to Execute Never (XN) memory regions.

BusFault

A BusFault is an exception that occurs because of a memory-related fault for an instruction or data memory transaction. This might be from an error that is detected on a bus in the memory system.

UsageFault

A UsageFault is an exception that occurs because of a fault that is related to instruction execution. This includes:

- An UNDEFINED instruction.
- An illegal unaligned access.
- Invalid state on instruction execution.

- An error on exception return.

The following can cause a UsageFault when the core is configured to report them:

- An unaligned address on word and halfword memory access.
- Division by zero.

SVCall

A Supervisor Call (SVC) is an exception that is triggered by the SVC instruction. In an OS environment, applications can use SVC instructions to access OS kernel functions and device drivers.

PendSV

PendSV is an interrupt-driven request for system-level service. In an OS environment, use PendSV for context switching when no other exception is active.

The SVCall and PendSV exceptions are banked between Secure and Non-secure states. When the SVC instruction is executed:

- If the processor is in Secure state, the SVC exception handling sequence fetches the exception vector from the Secure vector table and executes the SVCall handler in Secure state.
- If the processor is in Non-secure state, the SVC exception handling sequence fetches the exception vector from the Non-secure vector table and executes the SVCall handler in Non-secure state.

Similarly, the PendSVSet and PendSVClr bit in the Interrupt Control and State Register (ICSR) is banked. Secure software can also trigger a Non-secure PendSV using the Non-secure alias of ICSR (0xE002ED04).

The priority level registers for SVC and PendSV are also banked between Secure and Non-secure states.

SysTick

A SysTick exception is an exception the system timer generates when it reaches zero. Software can also generate a SysTick exception. In an OS environment, the processor can use this exception as system tick.

SysTick can exist in neither, either or both (banked) Security states.

Interrupt (IRQ)

An interrupt request, or IRQ, is an exception signaled by a peripheral, or generated by a software request. All interrupts are asynchronous to instruction execution. In the system, peripherals use interrupts to communicate with the processor.

Interrupts default to Secure state but can be retargeted to Non-secure state by Secure software.

SecureFault

For ARMv8‑M architecture with Main Extension the architecture defines an extra SecureFault exception. This exception is triggered by the various security checks that are performed. It is triggered, for example, when jumping from Non-secure code to an address in Secure code that is not marked as a valid entry point. Most systems choose to treat a SecureFault as a terminal condition that either halts or restarts the system. Any other handling of the SecureFault must be checked carefully to make sure that it does not inadvertently introduce a security vulnerability.

SecureFaults always target the Secure state.

3 Exception Properties

Each exception has an associated identification number, a vector address that is the exception entry point in memory, and a priority level which determines the order in which multiple pending exceptions are handled (the lower the priority number, the higher the priority level).

Exception property	Description
Vector address	Exception entry point in memory.
Priority level	Determines the order in which multiple pending exceptions are handled.
Exception number	Identification for the exception.

Each exception has an associated exception number.

Exception number	Name	Security
1	Reset	Secure
2	Non-Maskable Interrupt	Configurable
3	Secure HardFault	Secure
3	Non-secure Hard Fault	Non-secure
4	MemManage	Banked
5	BusFault	Configurable
6	UsageFault	Banked
11	SVCall	Banked
12	DebugMonitor	Configurable
14	PendSV	Banked
15	SysTick	Banked
16+N	Interrupts #0 -N	Configurable

3.1 Vector address

The vector table contains the reset value of the stack pointer and the start addresses, also called exception vectors, for all exception handlers.

The following figure shows the order of the exception vectors in the vector table. The least-significant bit of each vector must be 1, indicating that the exception handler is code.

Address		Exception #
0x40+4*N	External interrupt N	16+N
...	...	...
0x40	External interrupt 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	Reserved	13
0x30	Debug monitor	12
0x2C	SVC	11
0x28	Reserved	8-10
0x20	SecureFault*	7
0x1C	UsageFault	6
0x18	BusFault	5
0x14	MemManage	4
0x10	HardFault	3
0x0C	NMI	2
0x08	Reset	1
0x04	SP main	N/A
0x00		

*Reserved if Security Extension not implement

ARMv8-M architecture with Main extension

Exception #		Address
16+N	External interrupt N	0x40+4*N
...	...	...
16	External interrupt 0	0x40
15	SysTick*	0x3C
14	PendSV	0x38
12-13	Reserved (x2)	0x34 to 0x30
11	SVC	0x2C
4-10	Reserved	0x10 to 0x28
3	Hardfault	0x0C
2	NMI	0x08
1	Reset	0x04
N/A	SP main	0x00

*SysTick is optional

ARMv8-M architecture

The first entry contains the initial stack pointer. All other entries are the addresses for the exception handlers.

On system reset, the vector table is at an IMPLEMENTATION DEFINED address. Privileged software can write to the Vector Table Offset Register (VTOR) to relocate the vector table start address to a different memory location. It is IMPLEMENTATION DEFINED which bits are writable.

The silicon vendor must configure the top range value, which depends on the number of interrupts implemented. The minimum alignment is 32 words, enough for up to 16 interrupts. For more interrupts, adjust the alignment by rounding up to the next power of two. For example, if you require 21 interrupts, the alignment must be on a 64-word boundary because the required table size is 37 words, and the next power of two is 64.

ARM recommends that you locate the vector table in the CODE, SRAM, External RAM, or External Device areas of the system memory map.

Using the Peripheral, Private Peripheral Bus, or Vendor-specific memory areas can lead to unpredictable behavior in some systems. This is because the processor might use different interfaces for load/store instructions and vector fetch in these memory areas. If the vector table is located in a region of memory that is cacheable, you must treat any store to the vector as self-modifying code and use cache maintenance instructions to synchronize the update.

The Vector Table Offset Register

The VTOR specifies the base address of the vector table as an offset from address 0x0.

At reset the TBLOFF field is fixed at 0x00000000 unless an implementation includes configuration input signals that determine the reset value.

Setting up the vector table

The following example code sets up a vector table for an ARMv8-M architecture with Main Extension implementation, mapped to address 0x0 at reset:

```
…
AREA    RESET, DATA, READONLY, ALIGN=8
EXPORT _Vectors
EXPORT _Vectors_End
EXPORT _Vector_Size
_Vectors     DCD          _initial_sp
             DCD          Reset_Handler
             DCD          NMI_Handler         ; The vector table at boot is
                                              ; minimally required to have 4
; values: stack_top, reset
                                              ; routine location, NMI ISR
                                              ; location, Hardfault ISR
                                              ; location
             DCD          HardFault_handler
             DCD          MemManage_Handler
             DCD          BusFault_Handler
             DCD          UsageFault_Handler
             DCD          0, 0, 0, 0          ; The SVCall ISR location must be
                                              ; populated if the SVC instruction
                                              ; is used
             DCD          SVC_Handler
             DCD          DebugMon_Handler
             DCD          0
             DCD          PendSV_Handler
             DCD          SysTick_Handler     ; Once interrupts are enabled, the
                                              ; vector table must then contain
                                              ; pointers to all enabled (by mask)
                                              ; exceptions
             ; External Interrupts
; Add the vectors for the device specific external interrupts handler here
             DCD    <DeviceInterrupt>_IRQ_Handler ; 0: Default
_Vectors_End
```

```
_Vectors_Size  EQU  _Vectors_End - _Vectors
…
```

TheARMv8-M architecture table is very similar but MemManage, BusFault, Usage Fault and DebugMonitor do not exist and SysTick is optional.

3.2 Exception priorities

All exceptions have an associated priority, with a lower priority value indicating a higher priority to the exception, and configurable priorities for all exceptions except Reset, HardFault, and NMI. If software does not configure any priorities, all exceptions with a configurable priority have a priority of 0. Lower priority numbers take precedence.

Name	Exception Priority #
Interrupts #0 -N	0-255 (programmable)
SysTick	0-255 (programmable)
PendSV	0-255 (programmable)
DebugMonitor	0-255 (programmable)
SVCall	0-255 (programmable)
UsageFault	0-255 (programmable)
BusFault	0-255 (programmable)
MemManage	0-255 (programmable)
Non-secure Hard Fault	-1
Secure HardFault	-3 or -1 (programmable)
Non Maskable Interrupt	-2
Reset	-4

Note

- Configurable priority values are in the range 0-255. This means that the Reset, HardFault, and NMI exceptions, with fixed negative priority values, always have higher priority than any other exception.

- Reset, NMI and Hard Fault are the highest priority exceptions and the only exceptions with fixed priority. All others have settable priority.

For example, assigning a higher priority value to IRQ[0] and a lower priority value to IRQ[1] means that IRQ[1] has higher priority than IRQ[0]. If both IRQ[1] and IRQ[0] are asserted, IRQ[1] is processed before IRQ[0].

Note

For the ARMv8-M architecture, the available programmable priorities are limited to preempting priorities of 0, 64, 128 and 192. For the ARMv8-M architecture with Main Extension, the number of priority bits is between 3 and 8 inclusive, left-justified in an 8-bit field. However, the available range of preempting priorities is programmable and at most the top 7 bits, giving 0-254 in multiples of 2.

If multiple pending exceptions have the same priority, the pending exception with the lowest exception number takes precedence. For example, if both IRQ[0] and IRQ[1] are pending and have the same priority, then IRQ[0] is processed before IRQ[1].

4 Exception handlers

The processor handles exceptions using Interrupt Service Routines (ISRs), fault handlers, and system handlers.

Interrupt Service Routines	Handle interrupt exceptions for IRQ0-IRQ239.
Fault handlers	Handle fault exceptions for HardFault, MemManage fault, UsageFault, and BusFault.
System handlers	Handle system exceptions for NMI, PendSV, SVCall, SysTick, and system faults.

4.1 Writing exception handlers

Programmers should note the following points when writing exception handlers for the ARM®v8-M processor in C code or Assembly language.

The following general points should be noted when writing exception handlers:

- The ARMv8-M NVIC supports level-sensitive and pulse-sensitive IRQs. For interrupt sources that generate level-sensitive requests, clear the interrupt source. Some system handlers, such as SysTick, do not need to be cleared.
- Be careful of Main stack overflows. Take care with the additional stack requirements on the Secure Main stack in systems that support both Secure and Non-secure interrupts.
- There is no requirement to set the STKALIGN bit as in ARMv7-M. Bit[9] in the Configuration and Control Register is a RES1 bit.

The following points should be noted if writing exception handlers in C code:

- The ISR function must be type void and cannot accept arguments.
- The __irq keyword is optional but is recommended for clarity. For example,

```
__irq void SysTickHandler (void) { }
```

- No special assembly language entry or exit code is required.

The following points should be noted if writing exception handlers in Assembler language:

Handlers must manually save and restore any non-scratch registers which are used, such as R4R11, and the LR.

- Handlers must maintain 8-byte stack alignment if making function calls.
- Handlers must return from interrupt using EXC_RETURN with the proper instruction.
    - LDR PC
    - LDM and POP, which includes loading the PC
    - BX LR

4.2 Preemption

When the processor is executing an exception handler, an exception can preempt the exception handler if its priority is higher than the priority of the exception being handled.

Processor preemption is based around the execution Priority Level. To determine the current priority level, one has to check the IPSR register. If no interrupt is active (IPSR[ISR_NUMBER=0]), then the priority is the value stored in BASEPRI. If an interrupt is active, then the priority is determined by reading the priority bits of the active interrupt (IPSR[ISR_NUMBER]).

At reset the core has a priority level of MAX_LEVEL+1 (256). This means that any enabled interrupt (with even the lowest priority level) which is asserted will interrupt the core.

If an exception occurs with the same priority as the exception being handled, the handler is not preempted, irrespective of the exception number. However, the status of the new interrupt changes to pending.

When one exception preempts another the exceptions are called nested exceptions.

For example, consider the following exceptions where all have software configurable priority numbers:

- A has the highest priority.
- B has medium priority.
- C has lowest priority.

Example sequence of events:

1. Exception B occurs. The processor takes the exception and starts executing the handler for this exception. The execution priority is priority B.
2. Exception A occurs. Because its priority is higher than the execution priority, it preempts exception B and the processor starts executing the Exception A handler. Execution priority is now priority A.
3. Exception C occurs. Its priority is less than the execution priority so its status is pending.
4. The handler for exception A reduces the priority of exception A, to a priority lower than priority C. The execution priority falls to the highest priority of all active exceptions. This is priority B.

Exception C remains pending because its priority is lower than the current execution priority.

Only a pending exception with higher priority than priority B can preempt the current exception handler. Therefore, a new exception with lower priority than exception B cannot take precedence over the preempted exception B.

Late-arriving

The late-arriving mechanism speeds up preemption. If a higher priority exception occurs during state saving for a previous exception, the processor switches to handle the higher priority exception and initiates the vector fetch for that exception. It is IMPLEMENTATION DEFINED whether a processor does this.

There are several combinations of security states where the state saving might be affected by a late-arriving interrupt.

If a higher priority late-arriving Secure exception occurs during entry to a Non-secure exception when the Background Security state is Secure, it is IMPLEMENTATION DEFINED whether the stacking of the additional state context is rolled back.

The processor can accept a late arriving exception until the first instruction of the exception handler of the original exception enters the execute stage of the processor. On return from the exception handler of the late-arriving exception, the normal tail-chaining rules apply.

4.3 Tail-chaining

The ARMv8-M architecture tail-chaining mechanism speeds up exception servicing. On completion of an exception handler, if there is a pending exception that meets the requirements for exception entry, the stack pop is skipped and control transfers to the new exception handler.

4.4 Exception entry

Exception entry occurs when there is a pending exception with higher priority than the currently-executing context, in which case the new exception preempts this context.

Sufficient priority means that the exception has more priority than any limits set by the mask registers. An exception with less priority than this is pending but is not handled by the processor.

When the processor takes an exception, unless the exception is a tail-chained or a late-arriving exception, the processor pushes information onto the current stack. This operation is called stacking and the block of data written to the stack is referred as the stack frame. The basic stack frame contains eight words of data.

When using floating-point routines, or where an exception causes a change of security state, the processor might automatically stack additional registers to form an extended stack frame.

Note

Where stack space for floating-point state or security state is not allocated, the stack frame is the same as that of ARMv8-M implementations without an FPU.

Immediately after stacking, the stack pointer indicates the lowest address in the stack frame. ARMv8‑M requires that the stack frame is doubleword-aligned, and will automatically decrement the stack pointer if necessary to enforce this.

The stack frame includes the return address. This is the address of the next instruction in the interrupted program. This value is restored to the PC at exception return so that the interrupted program resumes.

In parallel to the stacking operation the processor performs a vector fetch that reads the exception handler start address from the vector table. When stacking is complete, the processor starts executing the exception handler. At the same time, the processor writes an EXC_RETURN value to the LR. This indicates which stack pointer corresponds to the stack frame and what operational mode the processor was in before the entry occurred.

If no higher priority exception occurs during exception entry, the processor starts executing the exception handler and automatically changes the status of the corresponding pending interrupt to active.

If another higher priority exception occurs during exception entry, the processor starts executing the exception handler for this exception and does not change the pending status of the earlier exception. This is the late arrival case.

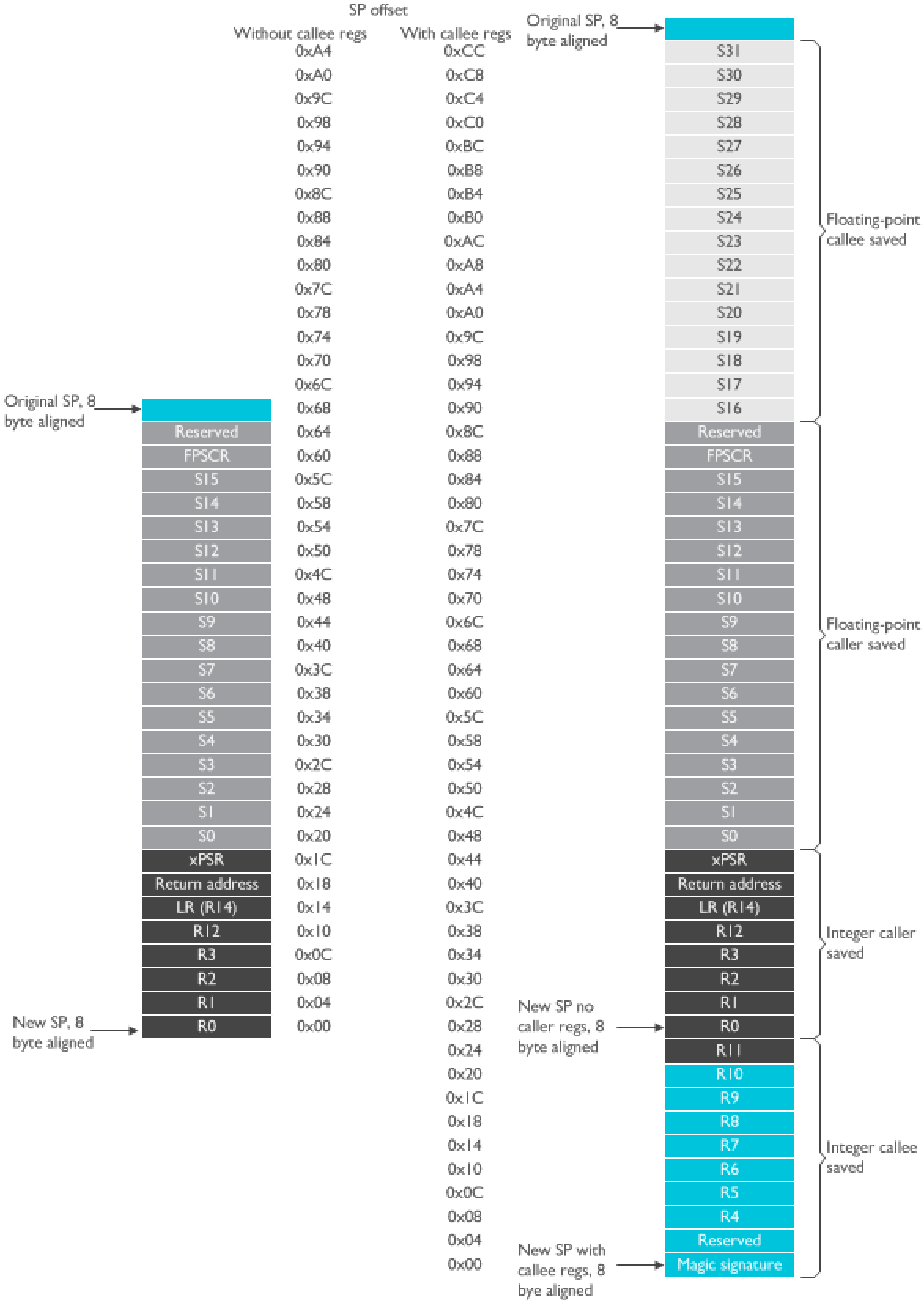
When an exception occurs the processor automatically transitions to the security state associated with that exception. This transition avoids the need for an SG instruction at the start of the Secure exception handlers, which would result in them being directly callable from the Non-secure state. If the exception and the exception vector target address are associated with different security states, a SecureFault is raised unless the exception is associated with the Non-secure state and targets an SG instruction. The Baseline profile has no SecureFault, so uses a HardFault exception instead.

4.5 Stack frame

For processors that do not implement the Security Extension, the stack frame format is divided into two sections, integer caller saved, and floating-point caller saved registers. To accommodate the additional state that must be preserved when an exception causes a transition from the Secure to the Non-secure state the format is extended.

The only mandatory section of the stack frame is the integer caller saved registers, with all other sections being optionally created depending on the state transitions being performed. ARMv8‑M-profile cores contain hardware to automatically stack and unstack the caller saved register state around exceptions, as defined in the ARM procedural calling standard. This reduces interrupt latency and enables extra optimizations like tail chaining. The approach taken by the ARMv8‑M architecture with Security Extension is to extend this mechanism to stack extra register contents if an exception causes a transition from the Secure to the Non-secure state. Some events also cause the processor to clear the registers. This prevents Secure data being visible to a Non-secure exception handler.

The following figure shows the structure of the stack frame:


Original SP, 8 byte aligned
New SP, 8 byte aligned
Original SP, 8 byte aligned
New SP no caller regs, 8 byte aligned
New SP with callee regs, 8 bye aligned
Floating-point callee saved
Floating-point caller saved
Integer caller saved
Integer callee saved

	SP offset		
	Without callee regs	With callee regs	
	0xA4	0xCC	S31
	0xA0	0xC8	S30
	0x9C	0xC4	S29
	0x98	0xC0	S28
	0x94	0xBC	S27
	0x90	0xB8	S26
	0x8C	0xB4	S25
	0x88	0xB0	S24
	0x84	0xAC	S23
	0x80	0xA8	S22
	0x7C	0xA4	S21
	0x78	0xA0	S20
	0x74	0x9C	S19
	0x70	0x98	S18
	0x6C	0x94	S17
	0x68	0x90	S16
Reserved	0x64	0x8C	Reserved
FPSCR	0x60	0x88	FPSCR
S15	0x5C	0x84	S15
S14	0x58	0x80	S14
S13	0x54	0x7C	S13
S12	0x50	0x78	S12
S11	0x4C	0x74	S11
S10	0x48	0x70	S10
S9	0x44	0x6C	S9
S8	0x40	0x68	S8
S7	0x3C	0x64	S7
S6	0x38	0x60	S6
S5	0x34	0x5C	S5
S4	0x30	0x58	S4
S3	0x2C	0x54	S3
S2	0x28	0x50	S2
S1	0x24	0x4C	S1
S0	0x20	0x48	S0
xPSR	0x1C	0x44	xPSR
Return address	0x18	0x40	Return address
LR (R14)	0x14	0x3C	LR (R14)
R12	0x10	0x38	R12
R3	0x0C	0x34	R3
R2	0x08	0x30	R2
R1	0x04	0x2C	R1
R0	0x00	0x28	R0
		0x24	R11
		0x20	R10
		0x1C	R9
		0x18	R8
		0x14	R7
		0x10	R6
		0x0C	R5
		0x08	R4
		0x04	Reserved
		0x00	Magic signature

Stack limit

Multiple nested interrupts or other techniques might be used to overflow the Secure stack, and therefore potentially overwrite other Secure data in memory.

To protect against this possibility, there are two stack limit registers, MSPLIM_S and PSPLIM_S. These registers limit the extent to which the Main and Process Secure stack pointers respectively can descend. Violation of the stack limit raises a synchronous UsageFault.

The ARMv8-M architecture with Main Extension also provides Non-secure stack limit registers.

4.6 Exception return

An exception return occurs when the processor is in Handler mode and executes a suitable instruction that loads the EXC_RETURN value into the PC.

An exception return also occurs when there is no pending exception with sufficient priority to be serviced and the completed exception handler was not handling a late-arriving exception.

The processor pops the stack and restores the processor state to the state it had before the interrupt occurred.

Any of the following instructions cause an exception return when the processor is in Handler mode:

- An LDM or POP instruction that loads the PC.
- An LDR instruction with PC as the destination.
- A BX instruction using any register.

Exception type	Value of ReturnAddress
NMI	Next instruction to be executed (Allows return after handling the NMI)
Hardfault (precise)	Instruction causing the fault (Usually a fatal error so return is unlikely)
Hardfault (imprecise)	Next instruction to be executed (Usually a fatal error so return is unlikely)
IRQ	Next instruction to be executed (Allows return after handling the IRQ)
SVC	Next instruction after the SVC instruction (allows return after handling the SVC)
SysTick (SysTick included)	Next instruction to be executed (Allows return after handling the SysTick)
PendSV	Next instruction to be executed (Allows return after handling the PendSV)

The processor can also return from an exception with the following instructions when the PC is loaded with the value `0xFFFF_FFxx` (EXC_RETURN). However, the value of `0xFFFF_FFxx` is an illegal address for execution (execute never memory).

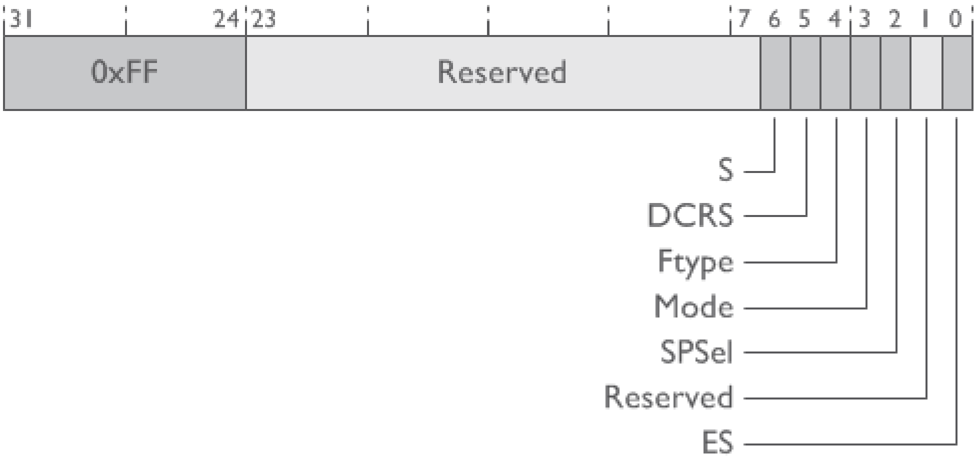
The core only uses this to identify that it is returning from an exception.

5 The EXC_RETURN register

The EXC_RETURN register is used to communicate additional information about which state to return to after handling an exception, and which registers need to be unstacked. EXC_RETURN is not an actual register that can be read.

On exception entry EXC_RETURN is the value that is loaded into the LR. The exception mechanism relies on this value to detect when the processor has completed an exception handler. The lowest 5 bits of this value provide information on the return stack and processor mode. The following table shows the EXC_RETURN values with a description of the exception return behavior.

The EXC_RETURN register has the following bit assignments:



Bits	Name	Description
[31:24]	0xFF	Prefix. Indicates that this is an EXC_RETURN value. This field reads as 0b11111111.
[7:23]	-	Reserved.
[6]	S	Indicates whether the stack frame to restore from is stored on a Secure or Non-secure stack. It is used together with the Mode bit and CONTROL.SPSEL to determine which of the four stack pointers is used to unstack the register state. The possible values of this bit are: 0 Non-secure stack used. 1 Secure stack entry.
[5]	DCRS	Default callee register stacking. Indicate whether the default stacking rules should apply, or whether the callee registers have already been pushed onto the stack and therefore do not need to be stacked again. The possible values of this bit are: 0 Stacking of the callee saved registers skipped. 1 Default rules for stacking the callee registers followed.

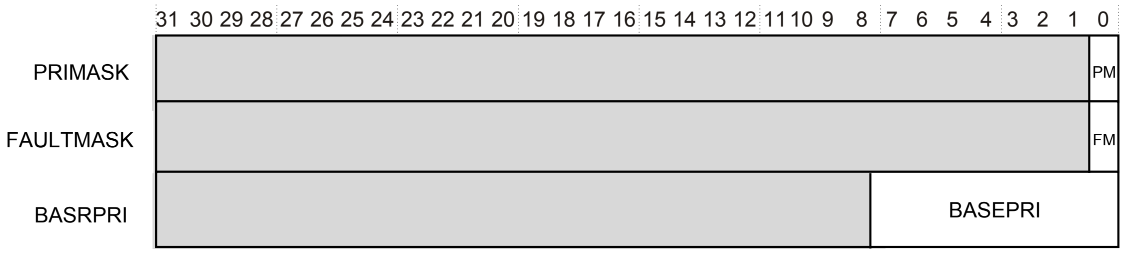
[4]	Ftype	Stack frame type. Indicates whether the stack frame is a standard integer-only stack frame, or an extended floating-point stack frame. The possible values of this bit are: 0 Extended stack frame. 1 Standard stack frame.
[3]	Mode	The Mode to return to (as in the existing ARMv7-M architecture). The possible values of this bit are: 0 Handler mode. 1 Thread mode.
[2]	SPSEL	Stack pointer selection. Indicates which stack pointer the exception frame resides on. The possible values of this bit are: 0 Main stack pointer. 1 Process stack pointer.
[1]	-	Reserved. Res1.
[0]	ES	Indicates the security domain the exception is taken to: The possible values of this bit are: 0 Non-secure. 1 Secure.

For implementations without the Security Extension;

EXC_RETURN	Condition
0xFFFFFFB0	Return to Handler mode
0xFFFFFFB8	Return to Thread mode using the main stack.
0xFFFFFFBC	Return to Thread mode using the process stack.

6 Special purpose mask registers

The exception mask registers disable the handling of exceptions by the processor. An example of the use of these registers would be to disable exceptions when they might affect timing critical tasks.



All exception mask registers can be modified using MRS and MSR instructions. PRIMASK and FAULTMASK can also be accessed using the CPS instruction.

The behavior differs depending on whether ARMv8-M architecture with Security Extension is implemented or not. All exceptions have an associated priority, with a low value indicating a high priority. Low priority exceptions take precedence.

Priority Mask Register

The PRIMASK register prevents servicing of all exceptions with configurable priority. Setting PRIMASK to 1 raises the execution priority to 0.

Fault Mask Register (only with ARMv8-M architecture with Main Extension)

The FAULTMASK register prevents servicing of all exceptions except Non-maskable Interrupts, HardFaults, or Resets.

Setting FAULTMASK to 1 raises the execution priority to -1, the priority of HardFault. Only privileged software executing at a priority below -1 can set FAULTMASK to 1. This means HardFault and NMI handlers cannot set FAULTMASK to 1. Returning from any exception except NMI clears FAULTMASK to 0.

PRIMASK and FAULTMASK map to the I and F flags.

```
CPSID I  ; set PRIMASK (disable interrupts)
CPSIE I  ; clear PRIMASK (enable interrupts)
CPSID F  ; set FAULTMASK (disable faults and interrupts)
CPSIE F  ; clear FAULTMASK (enable faults and interrupts)
```

Base Priority Mask Register (only with ARMv8-M architecture with Main Extension)

BASEPRI changes the priority level required for exception preemption. It has an effect only when BASEPRI has a lower value than the unmasked priority level of the currently executing software.

The number of implemented bits in BASEPRI is the same as the number of implemented bits in each field of the priority registers, and BASEPRI has the same format as those fields. A value of zero disables masking by BASEPRI.

When BASEPRI is set to a nonzero value, it prevents the servicing of all exceptions with the same or lower group priority level as the BASEPRI value.

BASEPRI is accessed like a Program Status Register

```
MSR{cond} BASEPRI, Rm     ; write BASEPRI
```

```
MRS{cond} Rd, BASEPRI     ; read BASEPRI
```

Alternatively, you can use the following CMSIS functions:

```
PRIMASK: __enable_irq(), __disable_irq()
```

```
FAULTMASK: __enable_fiq(), __disable_fiq()
```

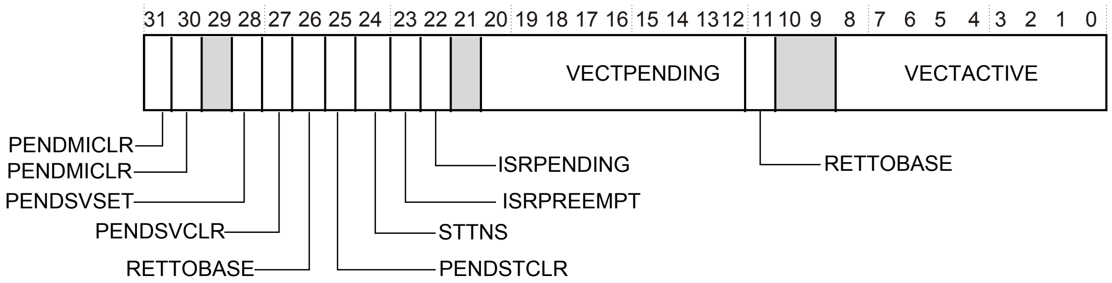
```
BASEPRI: __get_BASEPRI(), __set_BASEPRI(uint32_t value)
```

7 Other registers

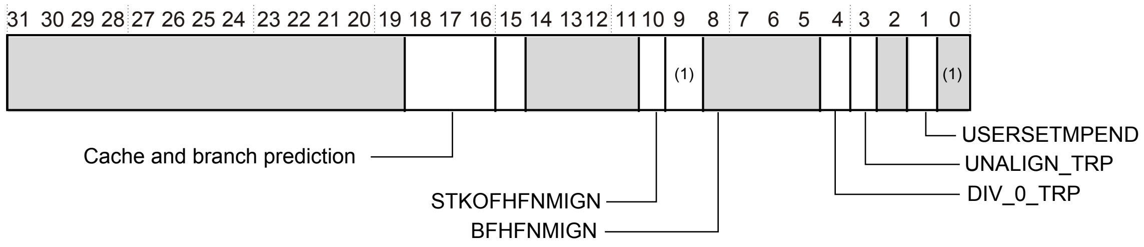
7.1 Interrupt Control and State Register

The ICSR is used to:

- Set to pending state, read pending state, or clear pending state of NMI, SVCall or SysTick.
- Control the security state of SysTick if a single SysTick is implemented with the Secure Extension.
- Indicate whether a pending exception is serviced on exit from Debug halt state (ISRPREEMPT).
- Indicate whether an external interrupt is pending (ISRPENDING).
- Indicate the exception number of the highest priority pending exception.
- Indicate whether there is only one active exception (RETTOBASE).
- Indicate the current active exception (same as IPSR).
- Set NMI to pending state.
- Indicate whether NMI is pending when in Debug state (ISRPENDING).
- Indicate whether a pending exception is serviced on exit from Debug halt state (ISRPREEMPT).
- Set and clear PENDSV and SysTick.
- Indicate the current active exception (same as IPSR)



7.2 Configuration and Control register



Bit value	Identifier	Description
[18]	BP	Branch prediction enable.
[17]	IC	Instruction cache enable.
[16]	DC	Data cache enable.
[10]	STKOFHFNMIGN	Determines the effect of stack overflow on HardFault and NMI handlers.
[8]	BFHFNMIGN	Determines the effect of synchronous data access faults on HardFault and NMI handlers.
[4]	DIV_0_TRP	Controls the trap for Divide by zero.
[3]	UNALIGN_TRP	Controls the trap of unaligned word and halfword accesses.
[1]	USERSETMPEND	Controls whether unprivileged software can access the Software Triggered Interrupt Register (STIR) which provides a mechanism for software to trigger an interrupt.