Fault handling and detection

Version 2.0

Revision Information

The following revisions have been made to this User Guide.

Date	Issue	Confidentiality	Change
07 July 2016	0100_00	Non-Confidential	First release
01 September 2016	0101_00	Non-Confidential	Second release
28 February 2017	0200_00	Non-Confidential	Third release

Product Status

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

Web Address

http://www.arm.com

Contents

1 Fault handling and detection

Error detection and correction techniques can be used to help mitigate the effect of errors in silicon devices. ARMv8-M processors include features that provide a means of detecting some of these errors.

In silicon devices, errors can occur because of:

- Software bugs.
- Usage errors, where the conditions are outside normal operational conditions. For example, temperature or supply voltage, or unexpected operations, such as invalid input data or operator errors.
- Memory corruptions, where stray radiation and other effects can cause the data that is stored in RAM to be corrupted.

Features of ARMv8‑M processors can enable software to manage or even correct some of the error conditions, and alert the users of the device to the event so that corrective or protective actions can be taken. Some of the ARMv8‑M devices are designed to detect more types of error conditions, and can handle the detected errors in a predictable manner, making them suitable for use in safety-related systems.

The features to detect and handle errors are divided into architectural features and implementation-specific features. The architecture for ARMv8‑M processors incorporates fault handling features by exceptions, and a Non-Maskable Interrupt (NMI) for handling system level errors, for example, brown out detection. Implementation-specific features such as Error Correcting Code (ECC) for memories are not covered here.

The ARMv8-M architecture is designed for devices with a small silicon footprint.

In a similar way to the ARMv6-M architecture, all fault events are considered as unrecoverable. There are no fault status registers in the ARMv8-M architecture, as there are in the ARMv8-M architecture with Main Extension. However, software developers can still analyze errors during software development using debug features like the Micro Trace Buffer (MTB) or Embedded Trace Macrocell (ETM). These features provide recent execution history, and therefore enable issues to be identified easily.

It is also possible for silicon chip designers create their own fault status registers and fault address registers to capture information about bus errors.

2 Fault exceptions

In ARMv8-M processors, error conditions are handled by fault exceptions. Fault exceptions are similar to interrupts, where piece of software that is called a handler is executed when fault event taken place.

Events can generate faults, for example:

- A bus error on:
    - An instruction fetch or vector table load.
    - A data access.
- An internally detected error like an UNDEFINED instruction.
- Attempting to execute an instruction from a memory region marked as Execute Never (XN).
- A privilege violation or an attempt to access an unmanaged region causing an MPU fault.

2.1 Description of fault types

Based on the processor implementation and the optional features included, there can be different type of fault exceptions available.

Fault type	Short name	Descriptions
Hard Fault	HardFault	Default Fault exception. Always enabled. Occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism.
Bus Fault	BusFault	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. Disabled by default. Only enable if necessary. Not available in the ARMv8-M architecture or ARMv6-M processors.
Usage Fault	UsageFault	Occurs because of a memory protection-related fault. Disabled by default. Only enable if necessary. The MPU or the fixed memory protection constraints determine this fault, for both instruction and data memory transactions. This fault is always used to abort instruction accesses to Execute Never (XN) memory regions. Not available in the ARMv8-M architecture or ARMv6-M

		processors.
Memory Management Fault	MemManage	Caused by violation of access permission set by the Memory Protection Unit (MPU) or attempting to execute code from XN (eXecute Never) address regions. Disabled by default. Only enable if necessary. Not available in the ARMv8-M architecture or ARMv6-M processors.
Secure Fault	SecureFault	Caused by Security violations. Non-secure branches to unauthorized Secure addresses cause a SecureFault exception. Available in the ARMv8-M architecture with main Extension only, and present only if the ARMv8-M Security Extension is implemented. In ARMv8-M architecture, security violations are handled by HardFault.

A simple system might only use the HardFault handler.

Note

- The Bus Fault, Usage Fault, Memory Management Fault, and Secure Fault exceptions are configurable fault exceptions. They can be enabled by software.
- Configurable Fault exceptions have programmable exception priority levels similar to interrupts and other system exceptions.
- The HardFault exception has a priority level of 1 (higher than all other exceptions apart from the NMI), or can be priority level 3 (higher than NMI) if it is triggered by a Security violation in Non-secure NMI in the ARMv8-M architecture.

If a fault event is triggered and the corresponding Configurable Fault exception is disabled, of if the priority level of the configurable fault exception is not high enough to trigger a preemption, the HardFault exception would be triggered instead. This is called escalation.

2.2 Fault escalation and HardFault

Fault escalation can occur for the following reasons:

- The fault is not enabled.
- The fault handler has insufficient priority to run.
- The fault handler encounters the same fault.

For example:

- An UNDEFINED instruction is decoded.
    - The processor tries to signal a UsageFault, but the UsageFault is not enabled.
- With the MPU enabled an interrupt handler makes and illegal memory accesss.

    - Processor tries to signal a memManage fault, bu the fault priority is lower than the priority of the running interrupt.

- An UNDEFINED instruction is encountered in a UsageFault handler.

    - An enabled fault handler causes the same kind of fault as is being serviced.

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.

The exception priority, and the values of the exception mask registers determine whether the processor enters the fault handler, and whether one fault handler can preempt another fault handler.

In some situations, a fault with configurable priority is treated as a HardFault. This 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 preempt 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 preempt 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.

If a Bus Fault occurs during a stack push when entering a Bus Fault handler, the Bus Fault does not escalate to a HardFault. This means that if a corrupted stack causes a fault, the fault handler executes even though the stack push for the handler failed. The fault handler operates but the stack contents are corrupted.

Note

- Only Reset and NMI can preempt the fixed priority HardFault. A HardFault can preempt any exception other than Reset, NMI, or another HardFault. In the ARMv8-M architecture, a Security violation in a Non-secure NMI handler (if BFHFNMINS is set to 1) would trigger and be preempted by a Secure HardFault exception.

- In the case where a bus error occurred during an exception vector fetch, this is always handled by HardFault exception and not BusFault.

3 Security state of fault exceptions

If the Security Extension is implemented in the processor, fault exceptions can target either the Secure state or Non-secure state. This is a change to the behavior in the ARMv6-M and ARMv7-M architectures.

Fault exception	Target state
HardFault	Default to Secure state. This can be configured to allow Non-secure HardFault by Secure software. However, security violations are still handled by the HardFault exception in Secure state.
BusFault	Default to Secure state. This can be configured to allow Non-secure BusFault by Secure software.
UsageFault	Can either be Secure or Non-secure state. Depends on the current state when the error event occurs.
MemManage	Can either be Secure or Non-secure state. Depends on the current state when the error event occurs.
SecureFault	Secure state only. Only available

A programmable bit in the *Application Interrupt and Reset Control Register* (AIRCR) called BFHFNMINS (BusFault, HardFault, and NMI Non-secure enable), bit [13] is used in the ARMv8-M architecture to permit Secure software to define whether fault exceptions and NMI are handled by Non-secure software.

The target state of the fault exception determines which vector table is used for fetching the exception vector and the execution state of the ISR. Because the vector tables for Secure and Non-secure software are separated, different fault handlers can be used when the fault events are targeted at different states.

4 Fault handling

In ARMv7-M and the ARMv8-M architecture with Main Extension, several *Fault Status Registers* (xFSR) are available to allow fault handlers to identify the cause of the fault exceptions. Each fault has an associated Fault Status Register. *Fault Address Registers* (XFAR) are available to indicate the address of the access that triggers the fault.

The fault status registers indicate the cause of a 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 shows the fault status and fault address registers.

Handler	Status register name	Address register name	Register description
HardFault	HFSR		HardFault Status Register
MemManage	MMFSR		MemManage Fault Status Register
		MMFAR	MemManage Fault Address Register
BusFault	BFSR		BusFault Status Register
		BFAR	BusFault Address Register
UsageFault	UFSR		UsageFault Status Register

The following table shows:

- The type of fault
- The handler that is used for the fault.
- The corresponding fault status register, and
- The register bit that indicates that the fault has occurred.

Fault	Handler	Bit name	Fault status register
Bus error on a vector read	HardFault	VECTTBL	HardFault Status Register
Fault escalated to a hard fault		FORCED	
MPU or default memory map mismatch:	MemManage	-	MemManage Fault Status Register
On instruction access		IACCVIOL	
On data access		DACCVIOL	
During exception stacking		MSTKERR	
During exception unstacking		MUNSKERR	
During lazy floating-point state preservation		MLSPERR	
Bus error:	BusFault	-	-
During exception stacking		STKERR	BusFault Status Register

During exception unstacking		UNSTKERR	
During instruction prefetch		BUSERR	
During lazy floating-point state preservation		LSPERR	
Precise data bus error		PRECISERR	
Imprecise data bus error		IMPRECISERR	
Attempt to access a coprocessor	UsageFault	NOCP	UsageFault Status Register
UNDEFINED instruction		UNDEFINSTR	
Attempt to enter an invalid instruction set state		INVSTATE	
Invalid EXC_RETURN value		INVPC	
Illegal unaligned load or store		UNALIGNED	
Divide By 0		DIVBYZERO	

Note

A fault inside a HardFault handler locks up the core.

4.1 Synchronous and Asynchronous bus faults

Bus Faults are subdivided into two classes:

A synchronous bus fault.	This is also described as a precise bus fault in older versions of ARM documents, and refers to the fault exception that takes place immediately after the bus transfer is carried out.
An asynchronous bus fault.	This is also described as an imprecise bus fault in older versions of ARM documents, and refers to the fault exception that can take place a certain time after the bus transfer is carried out, where the processor could have executed further instructions before the exception sequence started.

Synchronous bus fault

A synchronous BusFault can escalate into lockup if it occurs inside an NMI or HardFault handler. Cache maintenance operations can also trigger a bus fault.

The fault handler can use BFSR to determine whether faults are asynchronous or synchronous. Asynchronous faults are often unrecoverable, as you do not know which code caused the error.

Asynchronous bus fault

An asynchronous bus fault can happen when there is write buffering in the processor design. As a result, the processor pipeline proceeds to subsequent instruction execution before the bus error response is observed.

Debug accesses can also trigger Bus Faults. Debugger load or store accesses are synchronous, and are visible to the debugger interface only.

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 Bus Fault exception takes place. If the BusFault handler is not enabled, the HardFault exception is pended instead. The HardFault caused by the asynchronous BusFault never escalates into lockup.

4.2 Lockup state

The processor enters lockup state if a fault occurs when executing the NMI or HardFault handlers. When NMI is Non-secure and a Security violation is detected, it triggers a Secure HardFault at priority level 3. When the processor is in lockup state, it does not execute any instructions.

The processor remains in lockup state until either:

- It is reset.
- An NMI occurs.
- A debugger halts it.

Note

If lockup state occurs from the NMI handler, a subsequent NMI does not cause the processor to leave lockup state.

4.3 Non-secure fault handler considerations

When migrating software from ARMv6-M or ARMv7-M to ARMv8-M architecture, software developers that create Non-secure software must be aware of some possible changes:

- The HardFault and BusFault handler might no longer be used because the Secure software on chip has its own HardFault and BusFault handlers (AIRCR.BFHFNMINS might be set to 0).
- Self-reset might not be accessible from Non-secure software. When TrustZone technology for ARMv8-M is implemented, Secure software can configure whether Non-secure software can access the SYSRESETREQ bit in AIRCR. Also, the AIRCR.VECTRESET bit in ARMv7-M is not available in ARMv8-M architecture.

4.4 General considerations

In general, software developers for Secure software:

- Must always implement fault handlers to ensure that software developers creating Non-secure software can be aware of problems during software development.
- Secure software must utilize stack limit registers to ensure that stack overflow in Secure software can be detected and handled by a Secure HardFault or Usage Fault handler.
- Secure fault handlers must not call Non-secure functions to prevent Non-secure code gaining access to the Secure state which can be used for DoS attack.

5 AIRCR.BFHFNMINS

In ARMv8-M processors with ARM TrustZone technology for ARMv8-M implemented, the BFHFNMINS bit in the *Application Interrupt and Reset Control Register* (AIRCR) is available and is programmable by Secure privileged code only.

The possible values of this bit are:

0	BusFault, HardFault, and NMI are Secure.
1	BusFault and NMI are Non-secure and exceptions can target Non-secure HardFault.

It can be used in ARMv8-M devices where TrustZone technology is used for firmware protection. This is a common where software assets are preloaded and can be utilized by third-party software developers, but at the same time it must not be possible to reverse engineer the preloaded software.

If the application contains Secure libraries or a Secure RTOS, then the fault handling code interacts with the Secure code and the HardFault exception must be executed in Secure state. In such cases, AIRCR.BFHFNMINS must not be set to 1. If the code running in the Non-secure state needs to handle faults, then it should enable the individual Non-secure exceptions (UsageFault, MemManage Fault, etc for example) so that these faults can be handled by the Non-secure state and don't escalate to a Secure HardFault.

In applications where only a single security state is needed, it may be desirable to run in the Non-secure state and lock down access to the Secure state to prevent malicious root kits being installed by an attacker. In these situations, it may be desirable to set AIRCR.BFHFNMINS to 1 before locking down the Secure state so that HardFaults and NMI's can be handled by the Non-secure state.

Note

Software must clear the Bus Fault Address Register (BFAR) when setting AIRCR.BFHFNMINS to avoid exposure of the last fault address to the Non-secure state.

In some cases, when a Secure exception is triggered, the Secure handler may need to trigger the execution of Non-secure code. In cases where the original Secure exception is higher than the AIRCR.PRIS threshold (i.e. a lower numerical value), ARM does not recommend calling the Non-secure code directly from the Secure handler, as this could allow Non-secure software to gain a higher than expected execution priority level. Instead the handler can pend a Non-secure second-level exception handler so that the exception event can be handled by Non-secure software. ARM recommends that Secure exceptions only allow further execution of Non-secure code if the exception wasn't caused by an attempted attack.

Some of the fault handlers that are written for the ARMv6-M and ARMv7-M architecture contain code that extracts stacked program counters from the stack frame. Some of these handlers might need to be changed because the fault handler might be in the Secure world, while the stack frame is in the Non-secure stack, so the code to locate the stack frame must be changed.