Using Cortex-M3/M4/M7 Fault Exceptions

MDK Tutorial

AN209, Summer 2017, V 5.0

feedback@keil.com

Abstract

ARM® Cortex®-M processors implement an efficient exception model that traps illegal memory accesses and several incorrect program conditions. This application note describes the Cortex-M fault exceptions from the programmers view and explains their usage during the software development cycle. It also contains an example of a HardFault handler that reports information about the underlying fault.

Contents

Introduction

To detect problems as early as possible, all Cortex-M processors have a fault exception mechanism included. If a fault is detected, the corresponding fault exception is triggered and one of the fault exception handlers is executed. This application note describes the usage of fault exceptions. It lists the peripheral registers in the System Control Block (SCB) that control fault exceptions or provide information of their cause. Software examples show the usage of fault exceptions during program debugging or for recovering errors in the application software.

Prerequisites

This application note is accompanied by an example project with an exemplary HardFault handler. The example requires CMSIS version 5.0.0 or later to be present in your development environment. Download the example from www.keil.com/appnotes/docs/apnt_209.asp

Fault exception handlers

Fault exceptions trap illegal memory accesses and illegal program behavior. The following conditions are detected by fault exception handlers:

- **HardFault**: is the default exception and can be triggered because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism.
- **MemManage**: detects memory access violations to regions that are defined in the Memory Management Unit (MPU); for example, code execution from a memory region with read/write access only.
- **BusFault**: detects memory access errors on instruction fetch, data read/write, interrupt vector fetch, and register stacking (save/restore) on interrupt (entry/exit).
- **UsageFault**: detects execution of undefined instructions, unaligned memory access for load/store multiple. When enabled, divide-by-zero and other unaligned memory accesses are detected.

Fault exception numbers and priority

Each exception has an associated *exception number* and an associated *priority number*. To simplify the software layer, the CMSIS only uses IRQ numbers and therefore uses negative values for exceptions other than interrupts. The table lists fault exceptions ordered by their priority.

Exception	Exception Number	Priority	IRQ Number	Activation
HardFault	3	-1	-13	-
MemManage fault	4	Configurable	-12	Synchronous
BusFault	5	Configurable	-11	Synchronous when precise, asynchronous when imprecise.
UsageFault	6	Configurable	-10	Synchronous

The **HardFault** exception is always enabled and has a fixed priority (higher than other interrupts and exceptions, but lower than **Non-Maskable Interrupt NMI**). The **HardFault** exception is therefore executed in cases where a fault exception is disabled or when a fault occurs during the execution of a fault exception handler.

All other fault exceptions (**MemManage fault**, **BusFault**, and **UsageFault**) have a programmable priority. After reset, these exceptions are disabled and may be enabled in the system or application software using the registers in the System Control Block (SCB).

Priority escalation

Usually, the exception priority, together with the values of the exception mask registers, determines whether the processor enters the fault handler, and whether a 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 escalated to HardFault. Escalation to HardFault occurs when:

- A fault handler causes the same kind of fault as the one it is servicing. This escalation to HardFault occurs because a handler cannot preempt itself (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 BusFault occurs during a stack push when entering a BusFault handler, the BusFault 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.

Synchronous and asynchronous BusFaults

BusFaults are subdivided into two classes: synchronous and asynchronous bus faults. The fault handler can use the BFSR to determine whether faults are asynchronous (IMPRECISERR) or synchronous (PRECISERR).

Synchronous bus faults are also described as a precise bus faults. They refer to an exception that takes place immediately after the bus transfer is carried out. A synchronous BusFault can escalate into lockup if it occurs inside an NMI or HardFault handler. Cache maintenance operations can also trigger a BusFault.

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

Asynchronous bus faults are described as imprecise bus faults and can happen when there is write buffering in the processor design. As a result, the processor pipeline proceeds to the subsequent instruction execution before the bus error response is observed. 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, the HardFault exception is pended instead. A HardFault caused by an asynchronous BusFault never escalates into lockup. Asynchronous faults are often unrecoverable, as you do not know which code caused the error.

Fault types

The following table shows the fault type, fault handler, the fault status register, and the register bit name that indicates which fault has occurred. The bit names correlate with the fields shown in the µVision debug **Fault Reports** dialog. Details are further described in register description section.

Fault type	Handler	Status Register	Bit Name
Bus error on a vector read error	HardFault	HFSR	VECTTBL
Fault that is escalated to a hard fault			FORCED
Fault on breakpoint escalation			DEBUGEVT
Fault on instruction access	MemManage	MMFSR	IACCVIOL

Fault on direct data access			DACCVIOL
Context stacking, because of an MPU access violation			MSTKERR
Context unstacking, because of an MPU access violation			MUNSTKERR
During lazy floating-point state preservation			MLSPERR
During exception stacking	BusFault	BFSR	STKERR
During exception unstacking			UNSTKERR
During instruction prefetching, precise			IBUSERR
During lazy floating-point state preservation			LSPERR
Precise data access error, precise			PRECISERR
Imprecise data access error, imprecise			IMPRECISERR
Undefined instruction	UsageFault	UFSR	UNDEFINSTR
Attempt to enter an invalid instruction set state			INVSTATE
Failed integrity check on exception return			INVPC
Attempt to access a non-existing coprocessor			NOCPC
Illegal unaligned load or store			UNALIGNED
Stack overflow			STKOF
Divide By 0			DIVBYZERO

Fault exception registers

Control registers for fault exceptions

The *System Control Block* (SCB) provides system implementation information, and system control. This includes configuration, control, and reporting of the system exceptions. Some of its registers are used to control fault exceptions:

- The **Configuration and Control Register CCR** register controls the behavior of the **UsageFault** for divide-by-zero and unaligned memory accesses.
- The **System Handler Priority Registers SHP** control the exception priority.
- The **System Handler Control and State Register SHCSR** enables the system handlers, and indicates he pending status of the BusFault, MemManage fault, and SVC exceptions.

Address / Access	Register	Reset Value	Description
0xE000ED14 RW privileged	CCR	0x00000000	Configuration and Control Register: contains enable bits for trapping of divide-by-zero and unaligned accesses with the UsageFault.
0xE000ED18 RW privileged	SHP[12]	0x00	System Handler Priority registers: control the priority of exception handlers.
0xE000ED24 RW privileged	SHCSR	0x00000000	HardFault Status Register: contains bits that indicate the reason for HardFault

Configuration and Control Register CCR

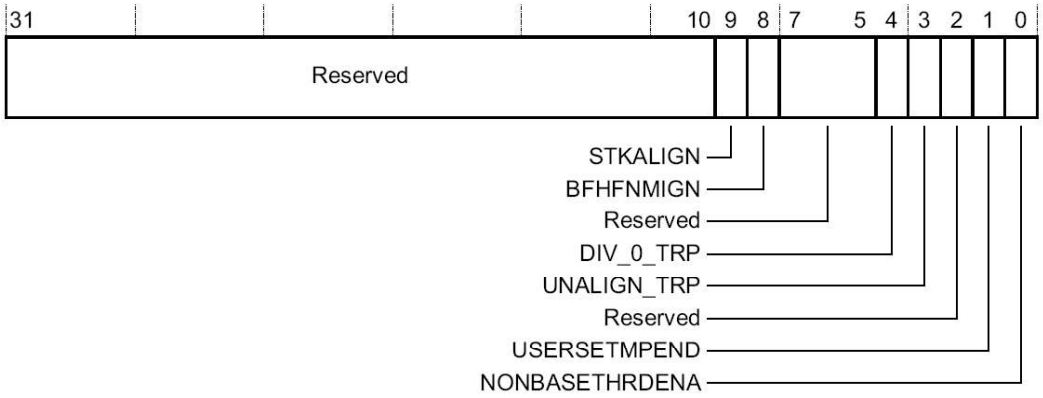

Figure 1 Bit assignments of the SCB->CCR register

The following bits of the **CCR** register control the behavior of the **Usage Fault**:

DIV_0_TRP: Enable **UsageFault** when the processor executes an SDIV or UDIV instruction with a divisor of 0:
0 = do not trap divide by 0; a divide by 0 returns a quotient of 0.
1 = trap divide by 0.

UNALIGN_TRP: Enable **UsageFault** when a memory access to unaligned addresses are performed:
0 = do not trap unaligned halfword and word accesses
1 = trap unaligned halfword and word accesses; an unaligned access generates a **UsageFault**. Note that unaligned accesses with LDM, STM, LDRD, and STRD instructions always generate a **UsageFault** even when UNALIGN_TRP is set to 0.

System Handler Priority Register SHP

The **SHP** registers set the priority level of exception handlers. The fault exceptions are controlled with:

SHP[0]: Priority level of the **Memory Management Fault**

SHP[1]: Priority level of the **BusFault**

SHP[2]: Priority level of the **UsageFault**

For programming interrupt and exception priorities CMSIS provides the functions **NVIC_SetPriority** and **NVIC_GetPriority**. The priority for the fault exceptions can be changed as shown below:

```
                :
NVIC_SetPriority (MemoryManagement_IRQn, 0x0F);
NVIC_SetPriority (BusFault_IRQn, 0x08);
NVIC_SetPriority (UsageFault_IRQn, 0x01);
                :
UsageFault_prio = NVIC_GetPriority (UsageFault_IRQn);
                :
```

System Handler Control and State Register SHCSR

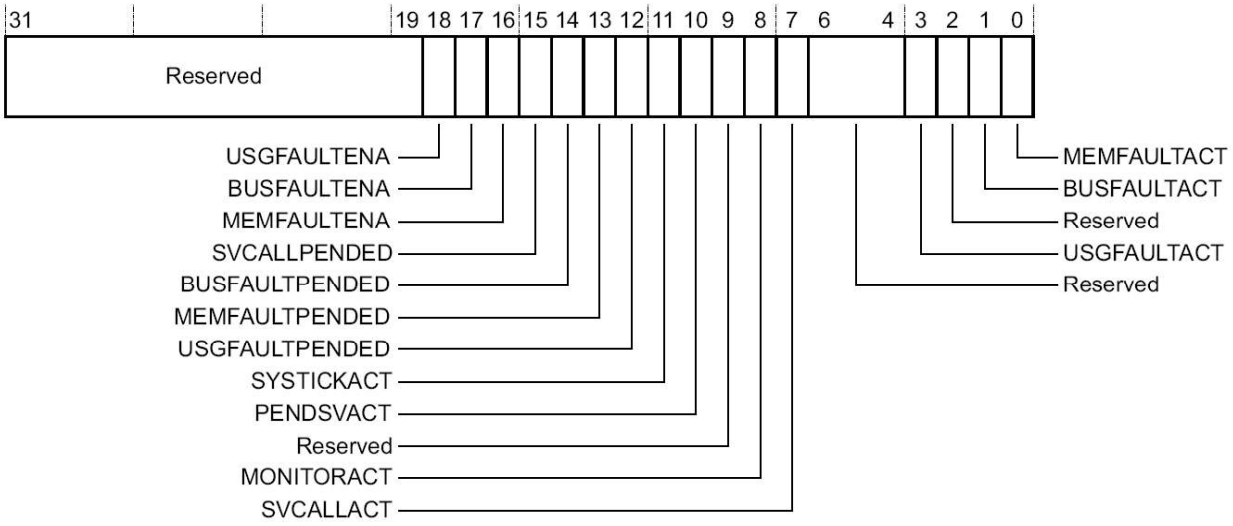

Figure 2 Bit assignments of the SCB->SHCSR register

The following bits of the **SHCSR** register belong to fault exceptions:

MEMFAULTACT: **Memory Management Fault** exception active bit, reads as 1 if exception is active.

BUSFAULTACT: **BusFault** exception active bit, reads as 1 if exception is active.

USGFAULTACT: **UsageFault** exception active bit, reads as 1 if exception is active.

USGFAULTPENDED: **UsageFault** exception pending bit, reads as 1 if exception is pending.

MEMFAULTPENDED: **Memory Management Fault** exception pending bit, reads as 1 if exception is pending.

BUSFAULTPENDED: **BusFault** exception pending bit, reads as 1 if exception is pending.

MEMFAULTENA: **Memory Management Fault** exception enable bit, set to 1 to enable; set to 0 to disable.

BUSFAULTENA: **BusFault** exception enable bit, set to 1 to enable; set to 0 to disable.

USGFAULTENA: **UsageFault** exception enable bit, set to 1 to enable; set to 0 to disable.

Although it is possible to write to all bits of the **SHCSR** register, in most software applications only a write to the enable bits makes sense. The Memory Management Fault, Bus Fault, and Usage Fault exceptions may be enabled with the following statement:

```
SCB->SHCSR |= SCB_SHCSR_USGFAULTENA_Msk
           |  SCB_SHCSR_BUSFAULTENA_Msk
           |  SCB_SHCSR_MEMFAULTENA_Msk; //enable Usage-/Bus-/MPU Fault
```

Status and address registers for fault exceptions

The table lists the names of fault status registers and fault address registers, showing the memory address of each register.

Handler	Status Register	Address Register	Address	Description
HardFault	HFSR		0xE000ED2C	HardFault Status Register
MemManage	MMFSR	MMFAR	0xE000ED28 0xE000ED34	MemManage Fault Status Register MemManage Fault Address Register
BusFault	BFSR	BFAR	0xE000ED29 0xE000ED38	BusFault Status Register BusFault Address Register
UsageFault	UFSR		0xE000ED2A	UsageFault Status Register
	AFSR		0xE000ED3C	Auxiliary Fault Status Register. Implementation defined content
	ABFSR		0xE000ED3C	Auxiliary BusFault Status Register. Only for Cortex-M7

HardFault Status Register HSFR Register

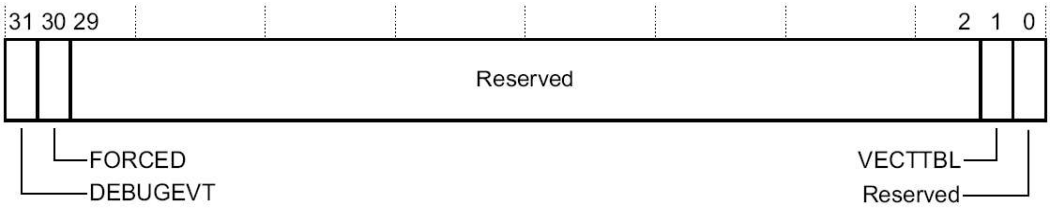

Figure 3 Bit assignments of the SCB->HSFR register

The Hard Fault status register indicates an incorrect usage of a CPU instruction and has following status bits:

VECTTBL: Indicates a Bus Fault on a vector table read during exception processing:
0 = no Bus Fault on vector table read
1 = Bus Fault on vector table read.
When this bit is set, the PC value stacked for the exception return points to the instruction that was preempted by the exception. This error is always a Hard Fault.

FORCED: Indicates a forced Hard Fault, generated by escalation of a fault with configurable priority that cannot be handled, either because of priority or because it is disabled:
0 = no forced Hard Fault
1 = forced Hard Fault.
When this bit is set, the Hard Fault handler must read the other fault status registers to find the cause of the fault.

DEBUGEVT: Reserved for Debug use. When writing to the register you must write 0 to this bit, otherwise behavior is unpredictable.

Configurable Fault Status Register CFSR Register

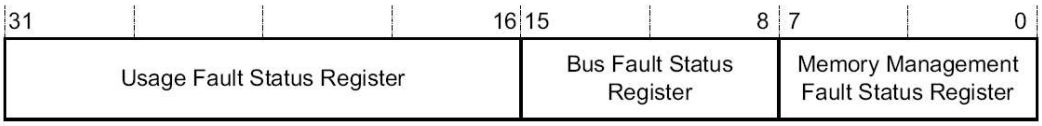

Figure 4 Bit assignments of the CFSR register

The CSFR register can be grouped into three status registers for: Usage Fault, Bus Fault, and Memory Management Fault.

MemManage Fault Status and Address Registers (MMFSR; MMFAR)

MemManage Status Register (MMFSR)

The MemManage fault status register (MMFSR) indicates a memory access violation detected by the Memory Protection Unit (MPU). Privileged access permitted only. Unprivileged accesses generate a BusFault.

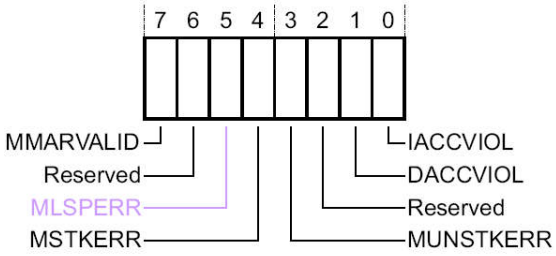

Figure 5 MMFSR bit assignment

MMFSR has following status bits:

IACCVIOL: Instruction access violation flag:
0 = no instruction access violation fault
1 = the processor attempted an instruction fetch from a location that does not permit execution. The PC value stacked for the exception return points to the faulting instruction. The processor has not written a fault address to the MMFAR. This fault condition occurs on any attempt of instruction fetches to an XN (eXecute Never) region, *even when the MPU is disabled or not present*. Potential reasons:
a) Branch to regions that are not defined in the MPU or defined as non-executable.
b) Invalid return due to corrupted stack content.
c) Incorrect entry in the exception vector table.

DACCVIOL: Data access violation flag:
0 = no data access violation fault
1 = the processor attempted a load or store at a location that does not permit the operation. The PC value stacked for the exception return points to the faulting instruction. The processor has loaded the MMFAR with the address of the attempted access.

MUNSTKERR: MemManage fault on unstacking for a return from exception:
0 = no unstacking fault
1 = unstacking for an exception return has caused one or more access violations.
This fault is chained to the handler which means that the original return stack is still present. The processor has not adjusted the SP from the failing return, and has not performed a new save. The processor has not written a fault address to the MMFAR. Potential reasons:
a) Stack pointer is corrupted
b) MPU region for the stack changed during execution of the exception handler.

MSTKERR: MemManage fault on stacking for exception entry:
0 = no stacking fault

1 = stacking for an exception entry has caused one or more access violations.
The SP is still adjusted but the values in the context area on the stack might be incorrect. The processor has not written a fault address to the MMFAR. Potential reasons:
a) Stack pointer is corrupted or not initialized
b) Stack is reaching a region not defined by the MPU as read/write memory.

MLSPERR: MemManage fault during floating point lazy state preservation (only Cortex-M4 with FPU):
0 = no fault occurred during floating-point lazy state preservation
1 = fault occurred during floating-point lazy state preservation

MMARVALID: MemManage Fault Address Register (MMFAR) valid flag:
0 = value in SCB->MMFAR is not a valid fault address
1 = SCB->MMFAR holds a valid fault address.
If a MemManage fault occurs and is escalated to a HardFault because of priority, the HardFault handler must set this bit to 0. This prevents problems on return to a stacked active MemManage fault handler whose SCB->MMFAR value has been overwritten.

MemManage Address Register (MMFAR)

The BFAR address is associated with a precise data access BusFault. Privileged access permitted only. Unprivileged accesses generate a BusFault.

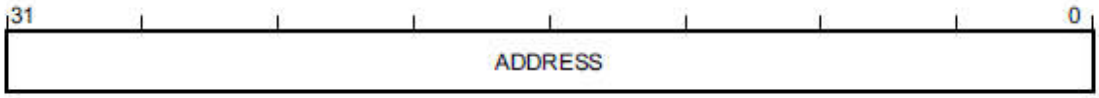

Figure 6 MMFAR bit assignment

ADDRESS: Data address for a MemManage fault. This register is updated with the address of a location that produced a MemManage fault. The MMFSR shows the cause of the fault. This field is valid only when MMFSR.MMARVALID is set. In implementations without unique BFAR and MMFAR registers, the value of this register is UNKNOWN if BFSR.BFARVALID is set.

BusFault Status and Address Register (BFSR; BFAR)

BusFault Status Register (BFSR)

The BusFault Status Register shows the status of bus errors resulting from instruction fetches and data accesses and indicates memory access faults detected during a bus operation. Only privileged access is permitted. Unprivileged access will generate a BusFault.

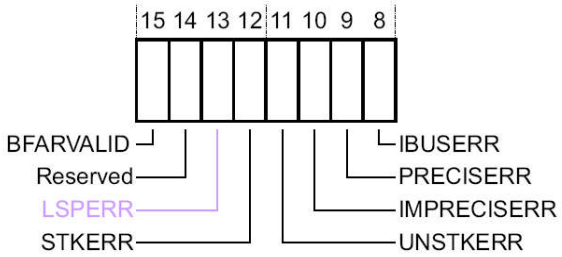

Figure 7 BFSR bit assignments

The BFSR status bits are:

IBUSERR: Instruction bus error. Records whether a BusFault on an instruction prefetch has occurred.
0 = no instruction bus error
1 = instruction bus error.
The processor detects the instruction bus error on prefetching an instruction, but it sets the

IBUSERR flag to 1 only if it attempts to issue the faulting instruction. When the processor sets this bit, it does not write a fault address to BFAR. Potential reasons:
a) Branch to invalid memory regions for example caused by incorrect function pointers.
b) Invalid return due to corrupted stack pointer or stack content.
c) Incorrect entry in the exception vector table.

PRECISERR: Precise data bus error:
0 = no precise data bus error
1 = a data bus error has occurred, and the PC value stacked for the exception return points to the instruction that caused the fault.
When the processor sets this bit, it writes the faulting address to BFAR.

IMPRECISERR: Imprecise data bus error:
0 = no imprecise data bus error
1 = a data bus error has occurred, but the return address in the stack frame is not related to the instruction that caused the error.
When the processor sets this bit it does not write a fault address to BFAR. This is an asynchronous fault. Therefore, if it is detected when the priority of the current process is higher than the BusFault priority, the BusFault becomes pending and becomes active only when the processor returns from all higher priority processes. If a precise fault occurs before the processor enters the handler for the imprecise BusFault, the handler detects both IMPRECISERR set to 1 and one of the precise fault status bits set to 1.

UNSTKERR: BusFault on unstacking for a return from exception:
0 = no unstacking fault
1 = unstack for an exception return has caused one or more BusFaults.
This fault is chained to the handler. This means that when the processor sets this bit, the original return stack is still present. The processor does not adjust the SP from the failing return, does not performed a new save, and does not write a fault address to BFAR.

STKERR: BusFault on stacking for exception entry:
0 = no stacking fault
1 = stacking for an exception entry has caused one or more BusFaults.
When the processor sets this bit, the SP is still adjusted but the values in the context area on the stack might be incorrect. The processor does not write a fault address to the BFAR. Potential reasons:
a) Stack pointer is corrupted or not initialized
b) Stack is reaching an undefined memory region.

LSPERR: BusFault during floating point lazy state preservation (only when FPU present):
0 = no fault occurred during floating-point lazy state preservation
1 = fault occurred during floating-point lazy state preservation

BFARVALID: BusFault Address Register (BFAR) valid flag:
0 = value in BFAR is not a valid fault address
1 = BFAR holds a valid fault address.
The processor sets this bit after a BusFault where the address is known. Other faults can set this bit to 0, such as a MemManage fault occurring later. If a BusFault occurs and is escalated to a HardFault because of priority, the HardFault handler must set this bit to 0. This prevents problems if returning to a stacked active BusFault handler who's BFAR value has been overwritten.

BusFault Address Register (BFAR)

The BFAR address is associated with a precise data access BusFault. Privileged access permitted only. Unprivileged accesses generate a BusFault.

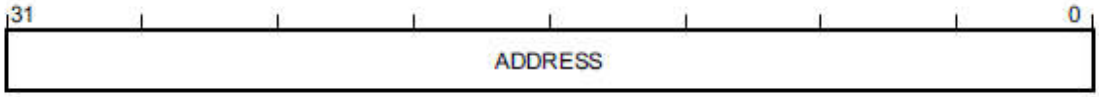

Figure 8 BFAR bit assignment

ADDRESS: Data address for a precise BusFault. This register is updated with the address of a location that produced a BusFault. The BFSR shows the reason for the fault. This field is valid only when BFSR.BFARVALID is set. In implementations without unique BFAR and MMFAR registers, the value of this register is UNKNOWN if MMFSR.MMARVALID is set.

UsageFault Status Register (UFSR)

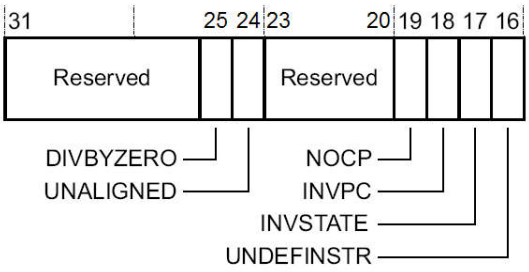

Figure 9 UFSR bit assignment

The **UsageFault Status Register UFSR** contains the status for some instruction execution faults, and for data access. Privileged access permitted only. Unprivileged accesses generate a BusFault.

The register has the following bits assigned:

UNDEFINSTR: Undefined instruction.
0 = no undefined instruction
1 = the processor has attempted to execute an undefined instruction.
When this bit is set, the PC value stacked for the exception return points to the undefined instruction. An undefined instruction is an instruction that the processor cannot decode.
Potential reasons:
a) Use of instructions not supported in the Cortex-M device.
b) Bad or corrupted memory contents.

INVSTATE: Invalid state:
0 = no invalid state
1 = the processor has attempted to execute an instruction that makes illegal use of the Execution Program Status Register (EPSR).
When this bit is set, the PC value stacked for the exception return points to the instruction that attempted the illegal use of the EPSR. Potential reasons:
a) Loading a branch target address to PC with LSB=0.
b) Stacked PSR corrupted during exception or interrupt handling.
c) Vector table contains a vector address with LSB=0.

INVPC: Invalid PC load UsageFault, caused by an invalid EXC_RETURN value:
0 = no invalid PC load
1 = the processor has attempted to load an illegal EXC_RETURN value to the PC as a result of an invalid context switch.
When this bit is set, the PC value stacked for the exception return points to the instruction that

tried to perform the illegal load of the PC. Potential reasons:
a) Invalid return due to corrupted stack pointer, link register (LR), or stack content.
b) ICI/IT bit in PSR invalid for an instruction.

NOCP: No coprocessor. The processor does not support coprocessor instructions:
0 = no UsageFault caused by attempting to access a coprocessor
1 = the processor has attempted to access a coprocessor that does not exist.

UNALIGNED: Unaligned access UsageFault:
0 = no unaligned access fault, or unaligned access trapping not enabled
1 = the processor has made an unaligned memory access.
Enable trapping of unaligned accesses by setting the UNALIGN_TRP bit in the CCR. Unaligned LDM, STM, LDRD, and STRD instructions always fault irrespective of the setting of UNALIGN_TRP bit.

DIVBYZERO: Divide by zero UsageFault:
0 = no divide by zero fault, or divide by zero trapping not enabled
1 = the processor has executed an SDIV or UDIV instruction with a divisor of 0.
When the processor sets this bit to 1, the PC value stacked for the exception return points to the instruction that performed the divide by zero. Enable trapping of divide by zero by setting the DIV_0_TRP bit in the CCR to 1.

Note that the bits of the UsageFault status register are sticky. This means, as one or more fault occurs, the associated bits are set to 1. A bit that is set to 1 is cleared to 0 only by writing 1 to that bit, or by a reset.

Auxiliary Bus Fault Status Register ABFSR Register (Cortex-M7 only)

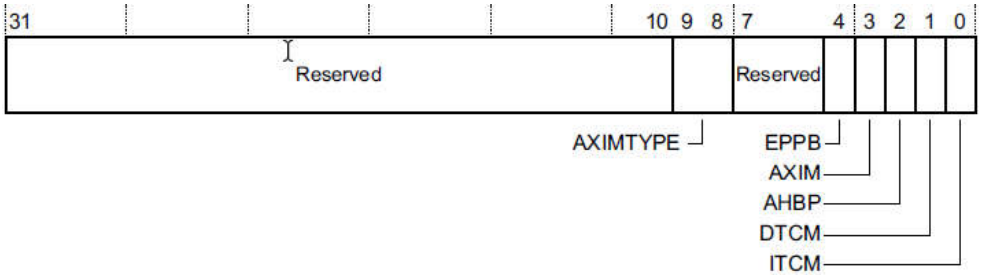

Figure 10 ABFSR bit assignment

The Auxiliary Bus Fault Status Register (ABFSR) stores information on the source of asynchronous bus faults. If a Bus Fault has occurred, the fault handler can read this register to determine which bus interface triggered the fault, and if the source is the AXIM interface, which error type is received. The ABFSR[4:0] fields remains valid until cleared by writing to the ABFSR with any value. The ASBFSR bit assignments are:

AXIMTYPE: Indicates the type of fault on the AXIM interface. The values are valid only when AXIM=1.
0b00 = OKAY
0b01 = EXOKAY
0b10 = SLVERR
0b11= DECERR

EPPB: Asynchronous fault on EPPB interface
AXIM: Asynchronous fault on AXIM interface
AHBP: Asynchronous fault on AHBP interface
DTCM: Asynchronous fault on DTCM interface
ITCM: Asynchronous fault on ITCM interface

Note: These interfaces might not be present on your implementation.

Implementing fault handlers

Fault handlers can be used in several ways. They can be used to shut-down the system safely, to inform users about the encountered problem, or they can trigger a self-test of the whole system.

A CMSIS compliant startup file (startup_*device*.s) defines all exception and interrupt vectors of a device. These vectors define the entry address of an exception or interrupt handler function. The following listing shows a typical vector table and the fault exception vectors are shown in blue.

```
__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
                DCD     HardFault_Handler         ; Hard Fault Handler
                DCD     MemManage_Handler         ; MPU Fault Handler
                DCD     BusFault_Handler          ; Bus Fault Handler
                DCD     UsageFault_Handler        ; Usage Fault Handler
                DCD     0                         ; Reserved
                  :
```

During debugging, a fault handler may be simply a BKPT (breakpoint) instruction which causes the debugger to stop. In general, all faults escalate to a HardFault, therefore it is sufficient to add the breakpoint instruction to the HardFault handler. When using MDK and a CMSIS-compliant device include file, you can overwrite the HardFault handler with your own code.

HardFault handler example

The AN209_Faults_and_Handler.zip contains an example project with an exemplary HardFault handler that saves the content of the core registers and displays it using `printf` statements. This custom handler consists of two parts:

- The handler wrapper is written in assembly language and it extracts the location of stack frame and passes it to the handler written in C as a pointer.
- The handler written in C language uses the information from the assembly wrapper to extract the relevant information and to display it using `printf` statements (refer to Figure 11 for an output example).

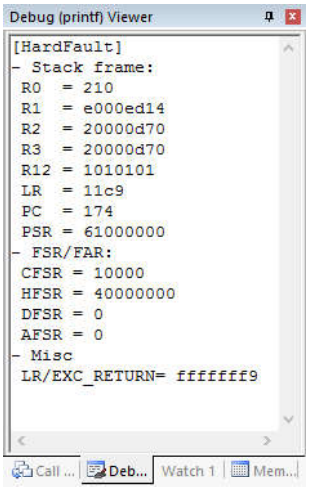

Figure 11 Handler output

For a final application, a fault handler may be implemented that performs:

- **System Reset**: by setting bit 2 (SYSRESETREQ) in AIRCR (Application Interrupt and Reset Control Register). This will reset most parts of the system apart from the debug logic. If you do not want to reset the whole system, just set the bit 0 (VECTRESET) in AIRCR which causes only a processor reset.
- **Recovery**: in some cases, it might be possible to resolve the problem that caused the fault exception. For example, in case of a coprocessor instruction, the handler may emulate the instruction in software.
- **Task termination**: for systems running a real-time operating system (RTOS), the task that created the fault may be terminated and restarted if needed.

Note: The following C statement is required in the initialization code to enable separate fault handlers for MemFault, BusFault, and UsageFault:

```
SCB->SHCSR |= SCB_SHCSR_USGFAULTENA_Msk
            |  SCB_SHCSR_BUSFAULTENA_Msk
            |  SCB_SHCSR_MEMFAULTENA_Msk; // enable Usage-/Bus-/MPU Fault
```

Fault handling considerations for ARM Cortex-M7

1. Cache maintenance operations can result in a BusFault. Such fault events are asynchronous (imprecise). This type of BusFault:
    a. does not cause escalation to HardFault where a BusFault handler is enabled.
    b. never causes lockup.

    Because the fault event is asynchronous, software code for cache maintenance operations must use memory barrier instructions, such as DSB, on completion, so that the fault event can be observed immediately.
2. The caches in Cortex-M7 support ECC (Error Correction Code). In the event of uncorrectable ECC errors, BusFault is triggered, and software can manage the ECC using Instruction Error Bank Register (IEBR0-1) and Data Error Bank Register (DEBR0-1).
3. Due to different design in the bus interface, there are some differences in behaviour of bus errors:
    a. The bus error in Cortex-M3 and Cortex-M4 is precise if the memory operation is a write to a strongly ordered memory region. On Cortex-M7, the same operation can trigger imprecise bus error.
    b. In Cortex-M3 and Cortex-M4, the exception sequence starts after outstanding buffered write is completed. So, if a bus error occurred, the stacked PC has the same context as the code that triggered the bus error. In Cortex-M7, the exception handling sequence can start before the write buffer is drained. In such case, the stacked PC could be showing a different context (for example, an IRQ handler that was triggered shortly after the buffered write took place). As a result, on Cortex-M7 the BusFault handler cannot rely on stacked PC to determine the fault location if the BusFault is asynchronous.

Debugging faults with µVision®

The example project (www.keil.com/appnotes/docs/apnt_209.asp) demonstrates three different fault exceptions:

- Calling an instruction that is not part of the ARM instruction set causes the undefined instruction UsageFault (UNDEFINSTR)
- Jumping to a valid memory address but without the Thumb bit set will lead to an invalid state UsageFault (INVSTATE)
- Dividing any given number by zero will trigger the division-by-zero UsageFault (DIVBYZERO)

Install the pack and copy the example project "AN209 Fault Exception Examples" to your local drive. Using the Configuration Wizard view of the project_configuration.h file, you can configure which fault exception to trigger. If you enable the HardFault handler, you will get useful information displayed in the Debug (printf) View window. Refer to the **HardFault handler example** on page 13 for more information.

Determining which exception has occurred

This example illustrates the behavior of a typical unexpected exception and shows how the µVision Debugger is used to analyze the cause of the problem. During execution, the application becomes unresponsive and appears to hang. The debugger is used to stop the application:

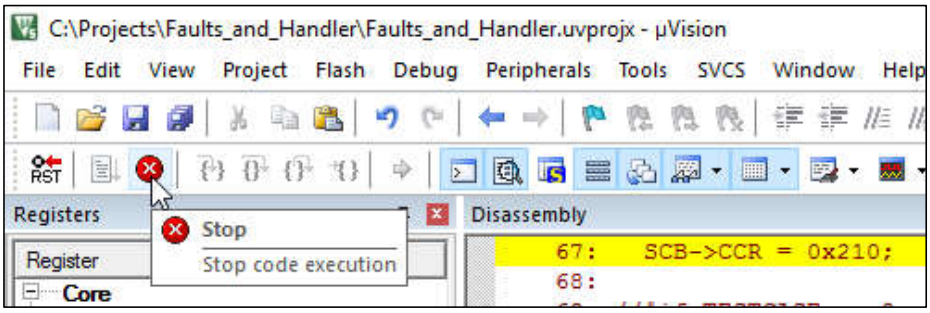

Figure 12 Stopping the processor in µVision

The application is in an infinite loop in the default Hard Fault handler, since there is no application-specific Hard Fault handler yet:

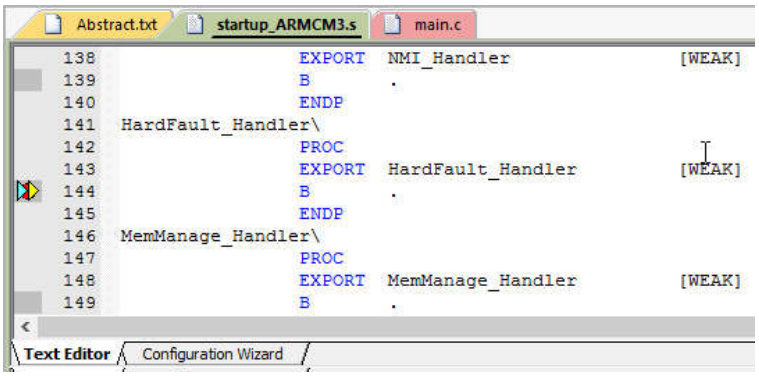

Figure 13 Branch-to-self in the HardFault handler

Accessing the Fault Reports dialog from the Peripherals menu

ARM Cortex-M fault registers will indicate exactly which exception has occurred. µVision provides the current values of all fault registers in the *Fault Reports* dialog available in the menu **Peripherals – Core Peripherals**:

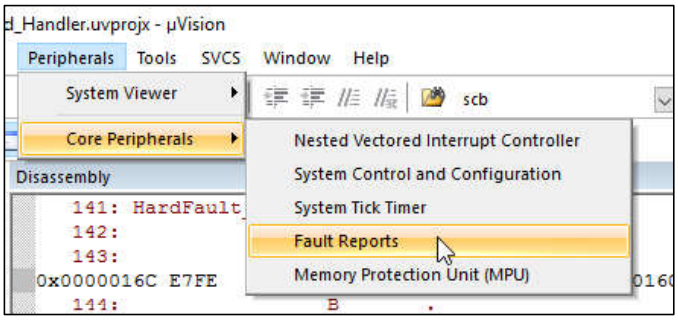

Figure 14 Opening the Fault Reports dialog

The **Fault Reports** dialog provides details of the exceptions that have occurred. In this case (TESTCASE INVSTATE in the pack's example project) it is an attempt to switch to an invalid state (ARM state) that caused a **Usage Fault** which was escalated to a **Hard Fault** as the Usage Fault handler is not enabled.

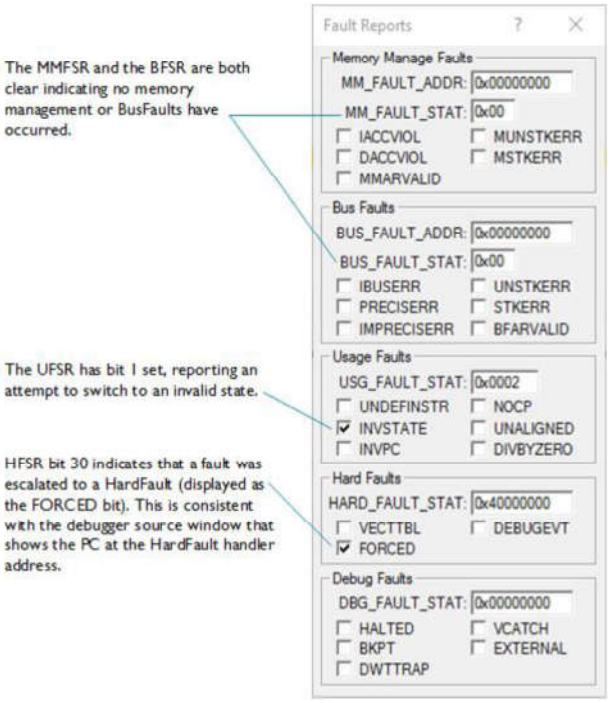

Figure 15 Faults Report Dialog

This **Fault Reports** dialog is a quick way to analyze a fault exception. If your debugger does not support such a dialog, the values may be reviewed using a memory window.

Determining where the exception has occurred

In the example project, enable the DIVBYZERO UsageFault and run the application until it ends up in a HardFault.

Right-click the *HardFault Handler* in the **Call Stack + Locals** window and select *Show Caller Code* to highlight the execution context at the point of occurrence:

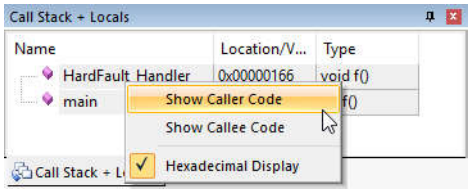

Figure 16 Call Stack used to display next scheduled instruction

Depending on the type of exception, the debugger will highlight the instruction that caused the exception or the instruction immediately after the one that caused the fault. This depends on whether or not the instruction causing the exception actually completed execution or not.

In main.c, the following line is highlighted:

```
i =i/j;    // i and j are 0 initialized -> Div/0
```

An exception saves the state of registers R0-R3, R12, PC & LR and either the MSP or the PSP (depending on the stack in use when the exception occurred). The current link register LR contains the EXC_RETURN value for the exception being serviced and this value indicates which stack holds the preserved register values from the application context. If bit 2 of the EXC_RETURN is zero then the main stack (MSP is saved) was used, else the process stack was used.

The *Registers* window provides access to the required information:

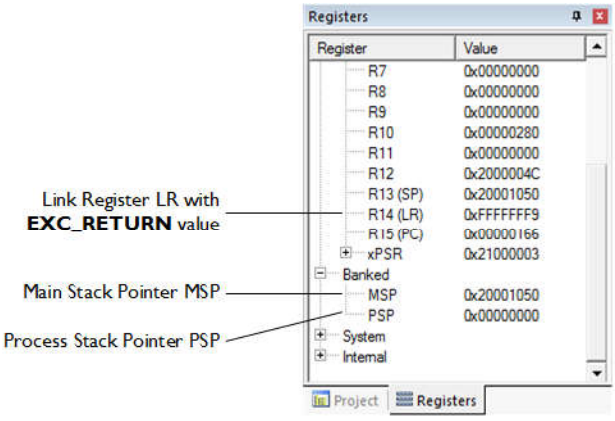

Figure 17 Registers Window

In this example, EXC_RETURN has the value

`0xfffffff9 = b_11111111111111111111111111111001 – bit 2 = 0`

which indicates the main stack contains the recently stored register values. The addresses of MSP and PSP are also visible here.

The MSP points to `0x20001050`. The memory window can be used to establish the previous execution context:

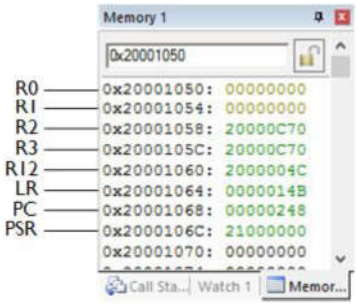

Figure 18 Execution context stored on the stack

This corresponds to the information we saw from the debugger earlier:

`PC = 0x00000248`	Next instruction that will cause the UsageFault (here: SDIV).
`R0-R3 & R12`	These are the values in the registers before the exception occurred

By using the details from the fault status registers and the appropriate stack the debugger provides the information needed to discover which exception has occurred and where. To further debug this particular problem the system must be reset with a watch point set on the function pointer that is being corrupted. This will reveal the root cause of the problem.

Further documentation

For information on the abort and exception model of the ARM Cortex-M3, Cortex-M4, or Cortex-M7, refer to the **Devices Generic User Guides** available from https://developer.arm.com/docs →Processors → Cortex-M.

A good reference book is: **The Definitive Guide to the ARM Cortex-M3 and Cortex-M4 Processors** by Joseph Yiu, ISBN 978-0-12-408082-9.

Revision history

- November 2013, 1.0:
  - Initial Version
- January 2016, 2.0:
  - System Handler Priority Register: corrected parameters of NVIC_SetPriority()
  - System Handler Control and State Register: corrected wrong SHCSR value 0x00070000 to 0x00007000
  - **Error! Reference source not found.**: changed __breakpoint (0) to __BKPT (0)
  - **Error! Reference source not found.**: corrected wrong SHCSR value 0x00070000 to 0x00007000
- March 2016, 3.0:
  - SCB->SHCSR on page 5 and 10 set to:
    SCB->SHCSR |= SCB_SHCSR_USGFAULTENA_Msk
    | SCB_SHCSR_BUSFAULTENA_Msk
    | SCB_SHCSR_MEMFAULTENA_Msk;
- September 2016, 4.0:
  - Corrected: PRECISEERR to PRECISERR
  - Corrected: Register window picture on page 14.
  - Adapted: Handling of hard faults from “Call Stack + Locals” window.
- August 2017, 5.0:
  - Application note revised completely, example project added.