Using Cortex-M3 and Cortex-M4 Fault Exception

Application Note 209

ARM KEIL Microcontroller Tools

Abstract

The Cortex-M processors implement an efficient exception model that also 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.

Contents

Introduction

Fault exceptions in the Cortex-M3 and Cortex-M4 processor trap illegal memory accesses and illegal program behavior. The following conditions are detected by fault exceptions:

- **Bus Fault**: detects memory access errors on instruction fetch, data read/write, interrupt vector fetch, and register stacking (save/restore) on interrupt (entry/exit).
- **Memory Management Fault**: 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.
- **Usage Fault**: detects execution of undefined instructions, unaligned memory access for load/store multiple. When enabled, divide-by-zero and other unaligned memory accesses are also detected.
- **Hard Fault**: is caused by Bus Fault, Memory Management Fault, or Usage Fault if their handler cannot be executed.

This application note describes the usage of this Cortex-M3 and Cortex-M4 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.

Further Documentation

A complete description of the exceptions is provided in the **Cortex-M3 Technical Reference Manual** or **Cortex-M4 Technical Reference Manual**. Both manuals are available at www.arm.com.

Another 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.

Cortex-M Fault Exceptions and Registers

A CMSIS compliant startup file (Startup_*device*) 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
                :
                :
```

The **Hard Fault** exception is always enabled and has a fixed priority (higher than other interrupts and exceptions, but lower than NMI). The **Hard Fault** 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 (**Memory Management Fault**, **Bus Fault**, and **Usage Fault**) 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).

Control Registers for Fault Exceptions

The **SCB->CCR** register controls the behavior of the **Usage Fault** for divide-by-zero and unaligned memory accesses.

Fault exceptions are enabled in the **SCB->SHCSR** register. If a fault exception is disabled and a related fault occurs, it is escalated to a **Hard Fault**. The **SCB->SHP** registers control the exception priority.

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

SCB->CCR Register

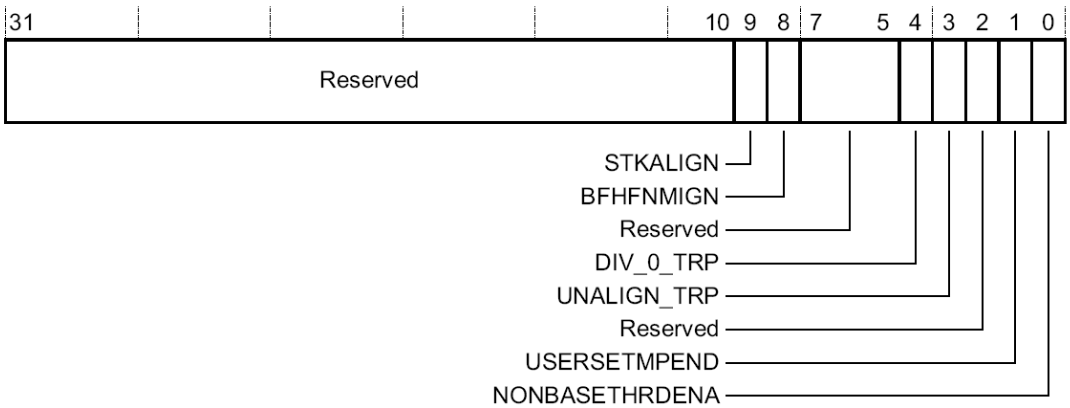

Figure 1: Bit assignments of the SCB->CCR register

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

DIV_0_TRP: Enable **Usage Fault** 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 **Usage Fault** 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 **Usage Fault**.
Note that unaligned accesses with LDM, STM, LDRD, and STRD instructions always generate a **Usage Fault** even when UNALIGN_TRP is set to 0.

SCB->SHP Registers

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

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

SCB->SHP[1]: Priority level of the **Bus Fault**

SCB->SHP[2]: Priority level of the **Usage Fault**

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);
                :
                :
```

SCB->SHCSR Register

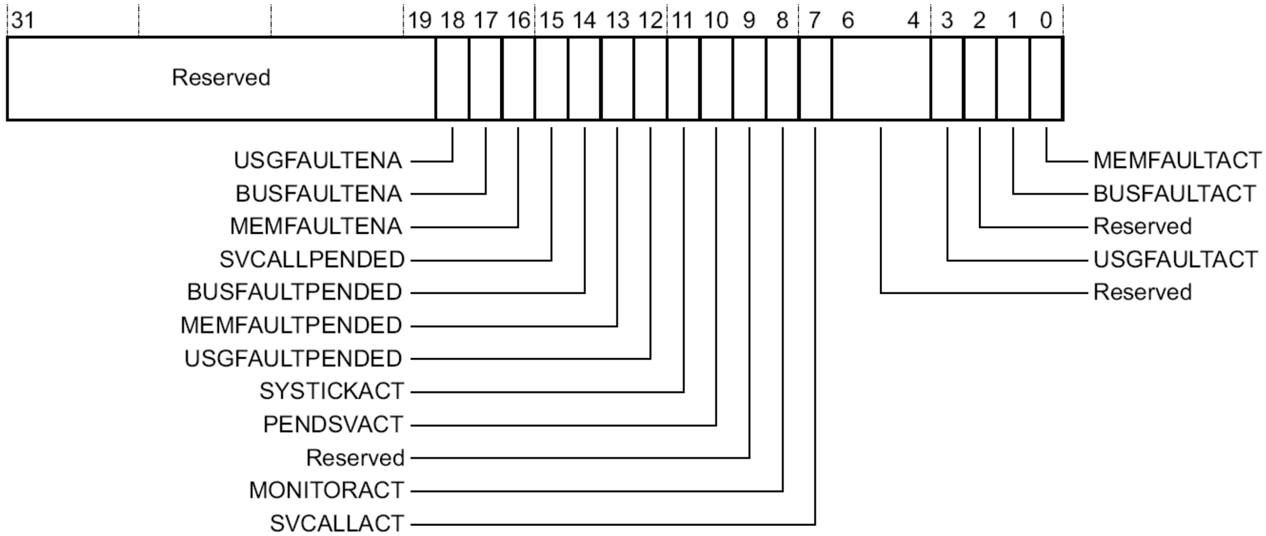

Figure 2: Bit assignments of the SCB->SHCSR register

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

MEMFAULTACT:	**Memory Management Fault** exception active bit, reads as 1 if exception is active.
BUSFAULTACT:	**Bus Fault** exception active bit, reads as 1 if exception is active.
USGFAULTACT:	**Usage Fault** exception active bit, reads as 1 if exception is active.
USGFAULTPENDED:	**Usage Fault** 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:	**Bus Fault** 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:	**Bus Fault** exception enable bit, set to 1 to enable; set to 0 to disable.
USGFAULTENA:	**Usage Fault** exception enable bit, set to 1 to enable; set to 0 to disable.

Although it is possible to write to all bits of the **SCB->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-, and MMU Fault
```

Status and Address Registers for Fault Exceptions

The fault status registers (SCB->CFSR and SCB->HFSR) and the fault address registers (SCB->MMAR and SCB->BFAR) contain detailed information about the fault and the memory address accessed.

Address / Access	Register	Reset Value	Description
0xE000ED28 RW privileged	SCB->CFSR	0x00000000	Configurable Fault Status Register: contains bits that indicate the reason for Memory Management Fault, Bus Fault, or Usage Fault
0xE000ED2C RW privileged	SCB->HFSR	0x00000000	Hard Fault Status Register: contains bits that indicate the reason for Hard Fault
0xE000ED34 RW privileged	SCB->MMFAR	Unknown	Memory Management Fault Address register: contains the memory address that caused the Memory Management Fault.
0xE000ED38 RW privileged	SCB->BFAR	Unknown	Bus Fault Address register: contains the memory address that caused the Bus Fault.

SCB->CFSR Register

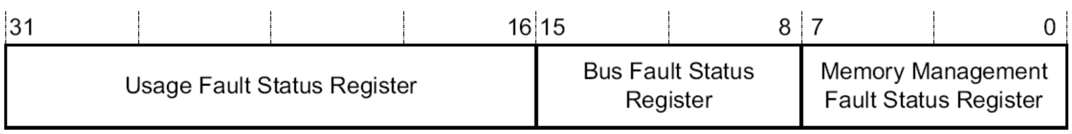

Figure 3: Bit assignments of the SCB->CFSR register

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

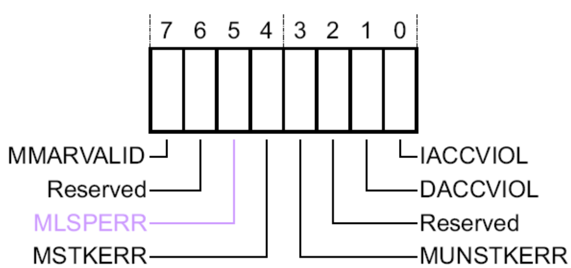

Figure 4: Bit assignments of the SCB->CFSR register – Memory Management Fault Status

The Memory Management Fault status register indicates a memory access violation detected by the Memory Protection Unit (MPU) and 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 MMAR. This fault condition occurs on any access 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 SCB->MMFAR with the address of the attempted access.

MUNSTKERR: Memory Management 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 SCB->MMFAR. Potential reasons:
a) Stack pointer is corrupted
b) MPU region for the stack changed during execution of the exception handler.

MSTKERR: Memory Management 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 SCB->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: Memory Management 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

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

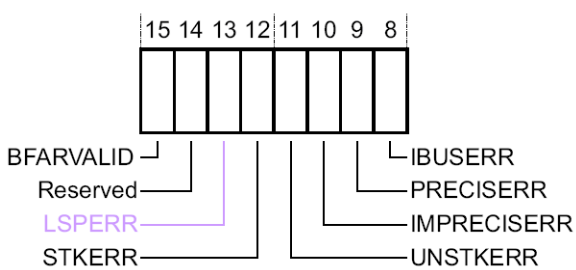

Figure 5: Bit assignments of the SCB->CFSR register – Bus Fault Status

The Bus Fault status register indicates a memory access fault detected during a BUS operation and has following status bits:

IBUSERR: Instruction bus error:
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 SCB->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 SCB->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 SCB->BFAR. This is an asynchronous fault. Therefore, if it is detected when the priority of the current process is higher than the Bus Fault priority, the Bus Fault 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 Bus Fault, 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 SCB->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: Bus 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

BFARVALID: Bus Fault Address Register (SCB->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 Bus Fault where the address is known. Other faults can set this bit to 0, such as a Memory Management Fault occurring later. If a Bus Fault occurs and is escalated to a Hard Fault because of priority, the Hard Fault handler must set this bit to 0. This prevents problems if returning to a stacked active Bus Fault handler whose SCB->BFAR value has been overwritten.

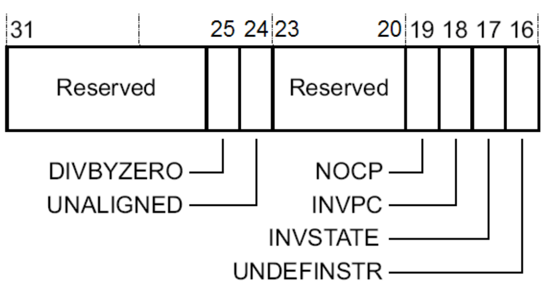

Figure 6: Bit assignments of the SCB->CFSR register – Usage Fault Status

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

UNDEFINSTR: Undefined instruction Usage Fault:
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 Usage Fault:
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 Usage Fault, caused by an invalid EXC_RETURN value:
0 = no invalid PC load
1 = the processor has attempted load of 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 Usage Fault. The processor does not support coprocessor instructions:
0 = no Usage Fault caused by attempting to access a coprocessor
1 = the processor has attempted to access a coprocessor that does not exist.

UNALIGNED: Unaligned access Usage Fault:
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 SCB->CCR. Unaligned LDM, STM, LDRD, and STRD instructions always fault irrespective of the setting of UNALIGN_TRP bit.

DIVBYZERO: Divide by zero Usage Fault:
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 SCB->CCR to 1.

Note that the bits of the Usage Fault 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.

SCB->HSFR Register

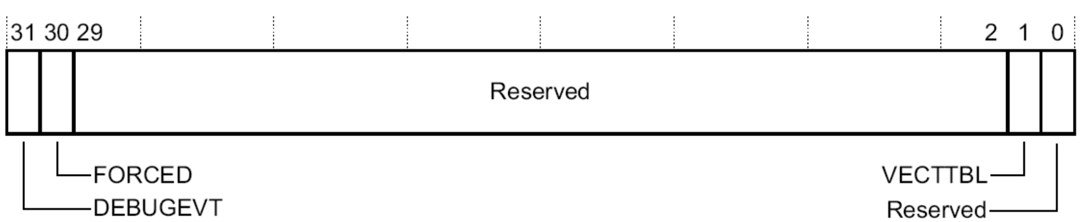

Figure 7: Bit assignments of the SCB->HSFR register

The Hard Fault status register indicates a 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.

SCB->MMFAR and SCB->BFAR Register

To determine which fault exception has occurred and what caused it you may examine these fault status registers.

The value of SCB->BFAR indicates the memory address that caused a Bus Fault and is valid if the bit BFARVALID in the SCB->CFSR register is set.

The value of SCB->MMFAR indicates the memory address that caused a Memory Management Fault and is valid if the bit MMFARVALID in the SCB->CFSR register is set.

Implementing Fault Handlers

During debugging, a Fault Handler may be simply a BKPT (breakpoint) instruction which causes the debugger to stop. By default all faults escalate to a Hard Fault, therefore it is sufficient to add the breakpoint instruction to the Hard Fault handler. When using MDK-ARM and a CMSIS-compliant device include file, you can overwrite the Hard Fault handler with the following C code. This code actually checks whether the system is connected to a debugger or not and may ship even in the end product.

```
void HardFault_Handler (void)  {
  if (CoreDebug->DHCSR & 1)  {      // check C_DEBUGEN == 1 -> Debugger Connected
    __BKPT (0);                     // halt program execution here
  }
  while (1);                        // enter endless loop otherwise
}
```

To trap **divide-by-zero** and **unaligned memory accesses**, the application initialization code should set the SCB->CCR register. This can be done with the following C statement:

```
  SCB->CCR |= 0x18;                 // enable div-by-0 and unaligned fault
```

For the final application a Fault Handler may be implemented that performs of the following:

- **System Reset**: by setting bit 2 (SYSRESETREQ) in SCB->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, you could just set the bit 0 (VECTRESET) in SCB->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 coprocessor instruction, the handler may emulate the instruction in software.
- **Task termination**: for systems that run a real-time operating system (RTOS), the task that created the fault may be terminated and restarted if needed.

Note, that the following C statement is required in the initialization code to enable separate fault handlers for Memory Management Fault, Bus Fault, and Usage Fault:

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

Debugging Faults with µVision

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:

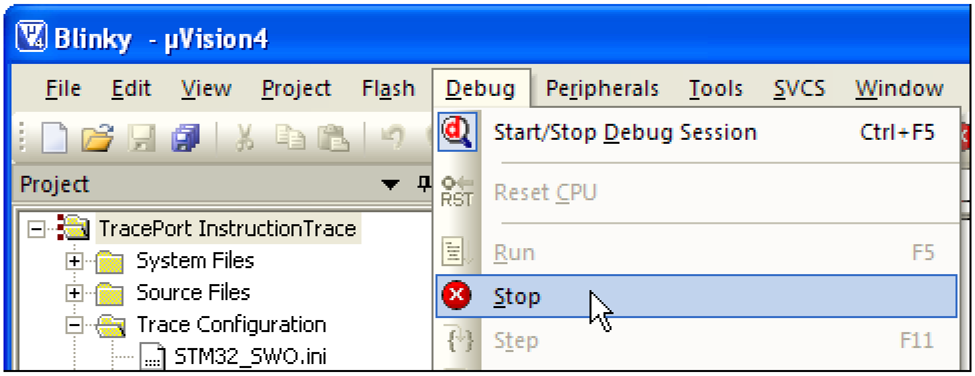

Figure 7: Stopping the processor in µVision

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

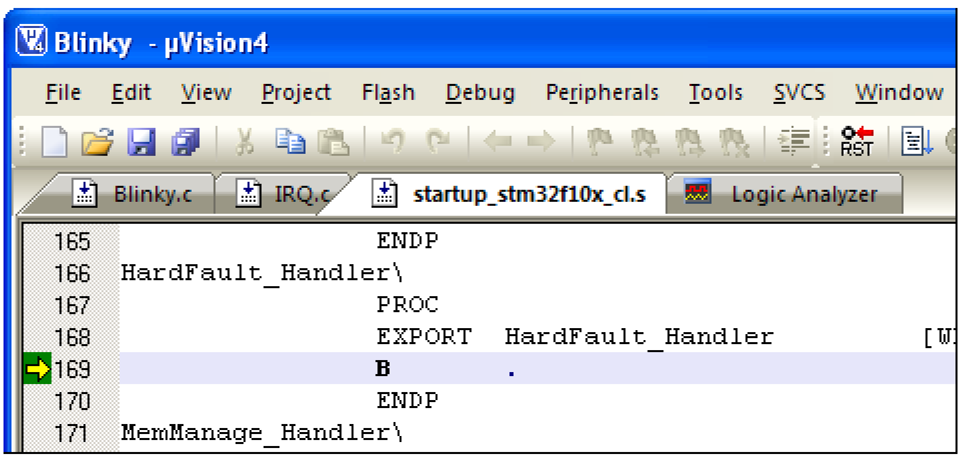

Figure 8: Branch-to-self Hard Fault handler

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

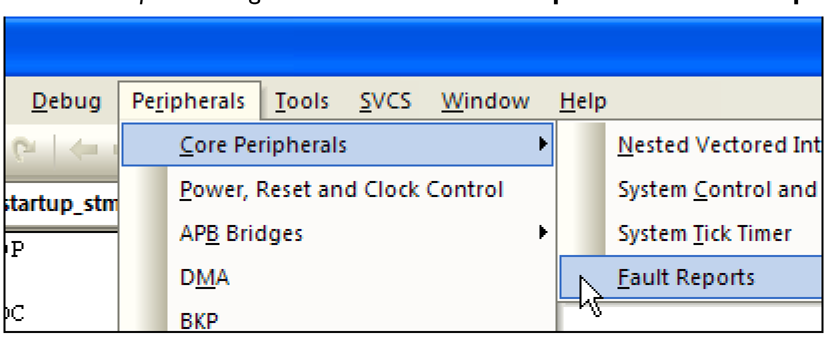

Figure 9: Opening the Fault Reports dialog

Accessing the Fault Reports dialog from the Peripherals menu

The **Fault Reports** dialog provides details of the exceptions that have occurred. In this case 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.

The MFSR and the BFSR are both clear indicating no memory management or bus faults have occurred.

The UFSR has bit 1 set reporting an attempt to switch to an invalid state.

The HFSR has bit 30 set indicating that the Usage Fault was escalated to a Hard Fault (displayed as the FORCED bit). This is consistent with the debugger source window that shows the PC at the Hard Fault handler address.

The DFSR has bit 0 set. This reflects the debug request signal asserted by the debugger which was used to halt the processor

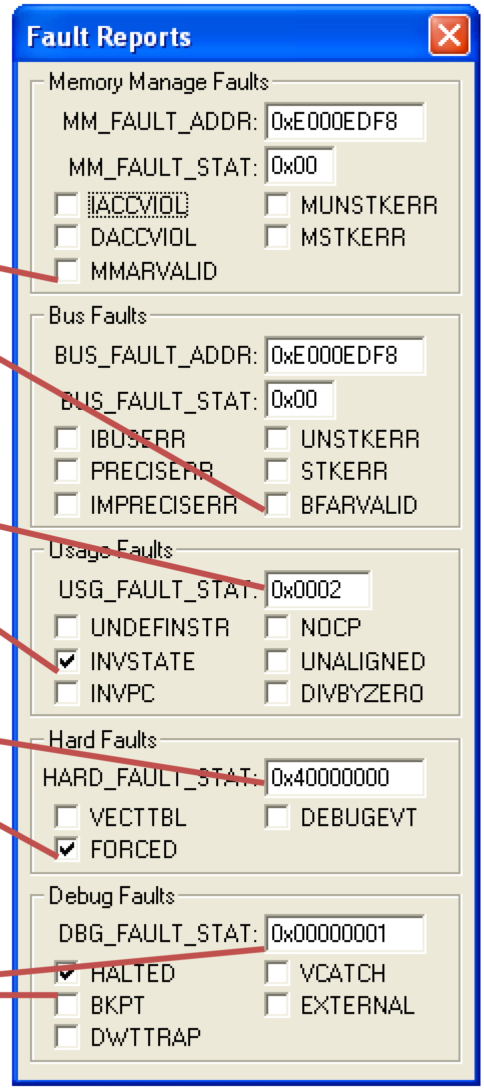

Figure 10 the Fault Reports dialog

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 occurred

The exception type has been identified as an attempt to switch to an invalid state. The debugger also provides the information needed to establish which instruction caused the exception to occur.

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:

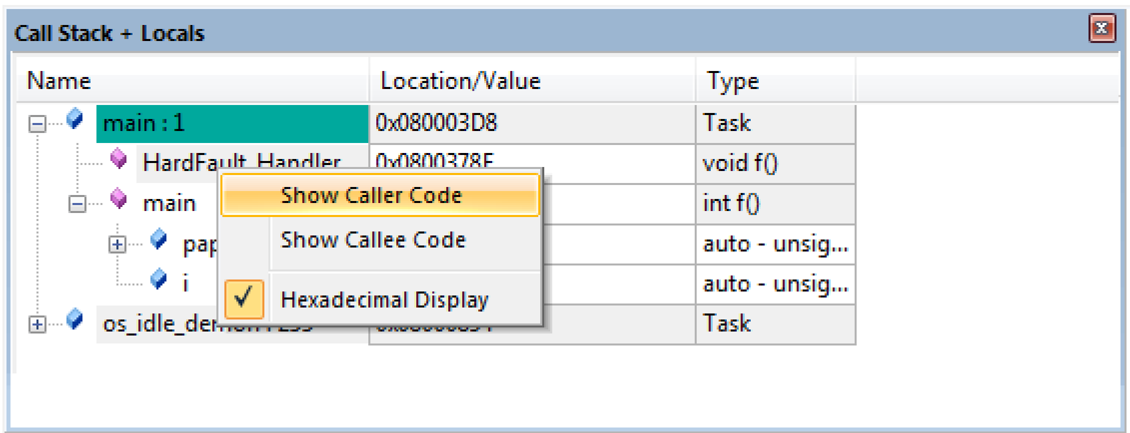

Figure 11 Call Stack used to display next scheduled instruction

Depending on the type of exception, the debugger will highlight the instruction that caused the exception (e.g. in the case of a Bus Fault) or the instruction immediately after the one that caused the fault (as in case of an attempt to change to an invalid state). This depends on whether or not the instruction causing the exception actually completed execution or not.

The source pane above highlights the compound C statement at line 213:

```
AD_val = (*funcArr[*AD_ptr/0x200])(*AD_ptr);
```

The disassembly shows the next instruction that is scheduled as:

```
MOV   r4,r0    at address      0x0800058A
```

This instruction will assign the return value of
`(*funcArr[*AD_ptr/0x200])(*AD_ptr) to AD_val.`
It is the previous instruction that caused the exception:

```
BLX   r1       at address      0x08000588
```

The Watch window can be used to determine the values of `*AD_ptr/0x200` and `funcArr` this will identify the address of the function that was called.

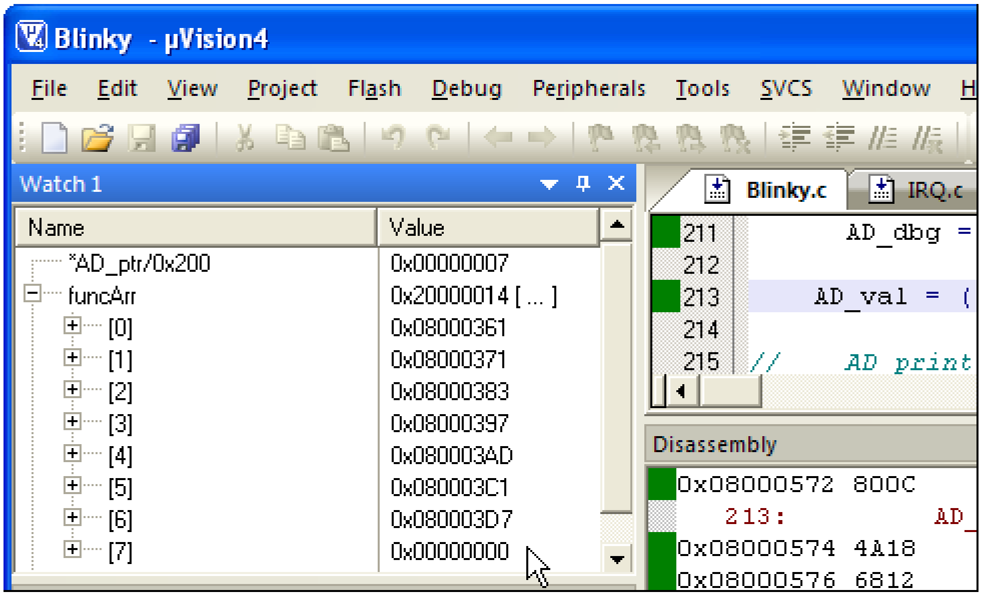

Figure 12 Watch window

This indicates that the previous instruction attempted to call `*funcArr[7]`. This resulted in a `BLX` to `0x00000000`. Any branch to an address with the least significant bit set to zero will attempt to switch the core to ARM state. This is invalid on a Cortex-M device and results in a Usage Fault exception with the INVSTATE bit being set in the SCB->UFSR. In this example the function pointer has been corrupted. The cause can be found by setting an access breakpoint on the array element and restarting the application.

An exception saves the state of registers R0-R3, R12, PC & LR either the Main Stack or the Process Stack (depends on the stack in use when the exception occurred). The current link register 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** was used
else (bit 2 of the **EXC_RETURN** is **one**) then the **Process Stack** was used

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

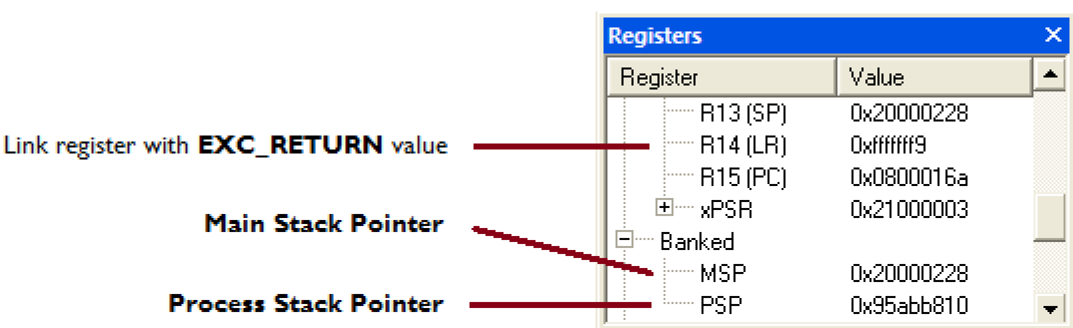

Figure 13 Register window

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 the Main Stack Pointer (MSP) and the Process Stack Pointer (PSP) are also visible here.

The MSP points to **`0x20000228`**. The memory window can be used to establish the previous execution context:

Execution context stored on the stack:

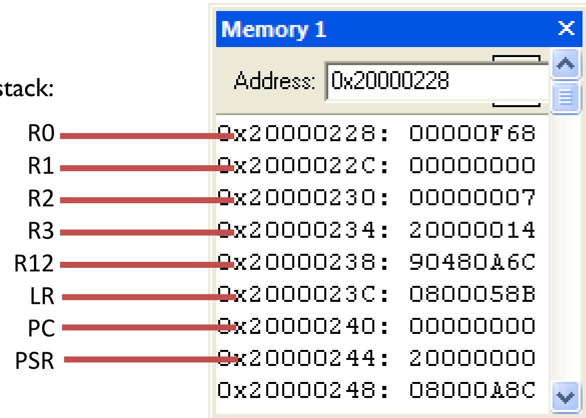

Figure 14 Contents of the Main Stack

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

`PC = 0x00000000`	The destination of the BLX r1 instruction. With bit 0 set to 0 this will attempt to switch to ARM state and is the cause of the exception.
`LR = 0x0800058B`	This is the return address from the **`BLX`** instruction. It corresponds to the instruction **`MOV r4,r0`** at address **`0x0800058A`** but has bit zero of the destination address set in order to ensure the processor stays in Thumb state.
`R0-R3 & R12`	These are the values in the registers before the exception occurred. **`R3`** has the address of **`FuncArr`**, **`R2`** has the array index and **`R1`** the address pointed to by **`FuncArr[7]`**.

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.

For further information on the abort and exception model of the Cortex-M3 please refer to the Cortex-M3 Technical reference Manual.

Revision History

- November 2013:
  - Initial Version
- January 2016:
  - SCB->SHP Registers: corrected parameters of NVIC_SetPriority()
  - SCB->SHCSR Register: corrected wrong SHCSR value 0x00070000 to 0x00007000
  - Implementing Fault Handlers: changed __breakpoint (0) to __BKPT (0)
  - Implementing Fault Handlers: corrected wrong SHCSR value 0x00070000 to 0x00007000
- March 2016:
  - 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:
  - Corrected: PRECISEERR to PRECISERR
  - Corrected: Register window picture on page 14.
  - Adapted: Handling of hard faults from “Call Stack + Locals” window.