CHAPTER

Fault Exceptions and Fault Handling

12

CHAPTER OUTLINE

The Definitive Guide to ARM® Cortex®-M3 and Cortex-M4 Processors. http://dx.doi.org/10.1016/B978-0-12-408082-9.00012-9

12.1 Overview of fault exceptions

Electronic systems can go wrong from time to time. The problems could be bugs in the software, but in many cases they can be caused by external factors such as:

- Unstable power supply
- Electrical noise (e.g., noise from power lines)
- Electromagnetic interference (EMI)
- Electrostatic discharge
- Extreme operation environment (e.g., temperature, mechanical vibrations)
- Wearing out of components (e.g., Flash/EEPROMs devices, crystal oscillators, capacitors) caused by repetitive programming or high-low temperature cycles
- Radiation (e.g., cosmic rays)
- Usage issues (e.g., end users didn't read the manual ☺) or invalid external data input

All these issues could lead to failure in the programs running on the processors. In many simple microcontrollers, you can find features like a watchdog timer and Brown-Out Detector (BOD). The watchdog can be programmed to trigger if the counter is not cleared within a certain time, and can be used to generate a reset or Non-Maskable Interrupt (NMI). The BOD can be used to generate a reset if the supply voltage drops to a certain critical level.

You can find a watchdog timer and BOD in many ARM® microcontrollers as well. However, when a failure occurs and the processor stops responding, it might take a bit of time for the watchdog to kick in. For most applications this is not a problem, but for some safety critical applications, a 1msec delay can be a matter of life or death.

In order to allow problems to be detected as early as possible, the Cortex®-M processors have a fault exception mechanism included. If a fault is detected, a fault exception is triggered and one of the fault exception handlers is executed.

By default, all the faults trigger the HardFault exception (exception type number 3). This fault exception is available on all Cortex-M processors including the Cortex-M0 and Cortex-M0+ processors. Cortex-M3 and Cortex-M4 processors have three additional configurable fault exception handlers:

- MemManage (Memory Management) Fault (exception type 4)
- Bus Fault (exception type 5)
- Usage Fault (exception type 6)

These exceptions are triggered if they are enabled, and if their priority is higher than the current exception priority level, as shown in Figure 12.1. These exceptions are called *configurable fault* exceptions, and have programmable exception priority levels (see section 7.9.5).

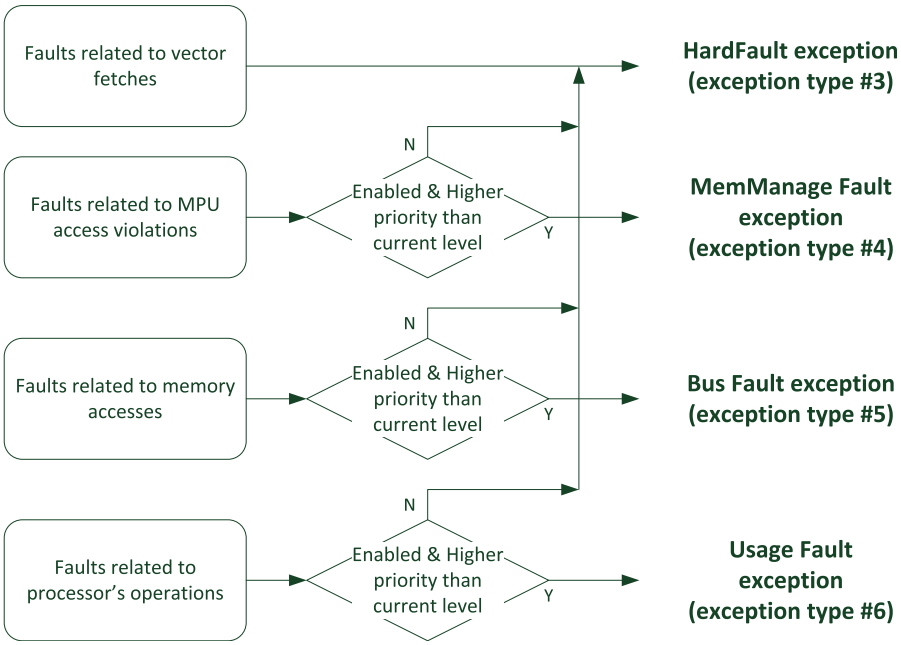

FIGURE 12.1

Fault exceptions available in ARMv7-M architecture

The fault handlers can be used in a number of ways:

- Shut down the system safely
- Inform users or other systems that it encountered a problem
- Carry out self-reset
- In the case of multi-tasking systems, the offending tasks could be terminated and restarted
- Other remedial actions can be carried out to try to fix the problem if possible (e.g., executing a floating point instruction with floating point unit turned off can cause an error, and can be easily solved by turning the floating point unit on)

Sometimes a system could carry out a number of different operations from the list above, depending on the type of fault detected.

To help detect what type of error was encountered in the fault handler, the Cortex-M3 and Cortex-M4 processors also have a number of Fault Status Registers (FSRs). The status bits inside these FSRs indicate the kind of fault detected. Although it might not pinpoint exactly when or where things went wrong, locating the source of the problem is made easier with these addition pieces of information. In addition, in some cases the faulting address is also captured by Fault Address Registers (FARs). More information about FSRs and FARs is given in section 12.4.

During software development, programming errors can also lead to fault exceptions. The information provided by the FARs can be very useful for software developers in identifying software issues in debugging.

The fault exception mechanism also allows applications to be debugged safely. For example, when developing a motor control system, you can shut down the motor by using the fault handlers before stopping the processor for debugging.

12.2 Causes of faults

12.2.1 Memory management (MemManage) faults

MemManage faults can be caused by violation of access rules defined by the MPU configurations. For example:

- Unprivileged tasks trying to access a memory region that is privileged access only
- Access to a memory location that is not defined by any defined MPU regions (except the Private Peripheral Bus (PPB), which is always accessible by privileged code)
- Writing to a memory location that is defined as read-only by the MPU

The accesses could be data accesses during program execution, program fetches, or stack operations during execution sequences. For instruction fetches that trigger a MemManage fault, the fault triggers only when the failed program location enters the execution stage.

For a MemManage fault triggered by stack operation during exception sequence:

- If the MemManage fault occurred during stack pushing in the exception entrance sequence, it is called a stacking error.
- If the MemManage fault occurrs during stack popping in the exception exit sequence, it is called an unstacking error.

The MemManage fault can also be triggered when trying to execute program code in eXecute Never (XN) regions such as the PERIPHERAL region, DEVICE region, or SYSTEM region (see section 6.9). This can happen even for Cortex®-M3 or Cortex-M4 processors without the optional MPU.

12.2.2 Bus faults

The bus faults can be triggered by error responses received from the processor bus interface during a memory access; for example:

- Instruction fetch (read), also called prefetch abort in traditional ARM® processors
- Data read or data write, also called data abort in in traditional ARM processors

In addition, the bus fault can also occur during stacking and unstacking of the exception handling sequence:

- If the bus error occurred during stack pushing in the exception entrance sequence, it is called a stacking error.
- If the bus error occurred during stack popping in the exception exit sequence, it is called an unstacking error.

If the bus error happened in the instruction fetch, the bus fault triggers only when the failed program location enters the execution stage. (A branch shadow access that triggers a bus error does not trigger the bus fault exception if the instruction does not enter execution stage.)

Please note that if a bus error is returned at vector fetch, the HardFault exception would be activated even when Bus Fault exception is enabled.

A memory system can return error responses if:

- The processor attempts to access an invalid memory location. In this case, the transfer is sent to a module in the bus system called default slave. The default slave returns an error response and triggers the bus fault exception in the processor.
- The device is not ready to accept a transfer (e.g., trying to access DRAM without initializing the DRAM controller might trigger the bus error. This behavior is device-specific.)
- The bus slave receiving the transfer request returns an error response. For example, it might happen if the transfer type/size is not supported by the bus slave, or if the peripherals determined that the operation carried out is not allowed.
- Unprivileged access to the Private Peripheral Bus (PPB) that violates the default memory access permission (see section 6.8).

Bus faults can be classified as:

- Precise bus faults − fault exceptions happened immediately when the memory access instruction is executed.
- Imprecise bus faults − fault exceptions happened sometime after the memory access instruction is executed.

The reason for a bus fault to become imprecise is due to the presence of write buffers in the processor bus interface (Figure 6.17). When the processor writes data to a bufferable address (see section 6.9 on memory access attributes, and section 11.2.5 on MPU Base Region attribute and Size register), the processor can proceed to execute the next instruction even if the transfer takes a number of clock cycles to complete.

The write buffer allows the processor to have higher performance, but this can make debugging a bit harder because by the time the bus fault exception is triggered, the processor could have executed a number of instructions, including branch instructions. If the branch target can be reached via several paths (Figure 9.16), it could be hard to tell where the faulting memory access took place unless you have an instruction trace (see Chapter 14, section 14.3.5). To help with debugging such situations, you can disable the write buffer using the DISDEFWBUF bit in the Auxiliary Control register (section 9.9).

Read operations and accesses to the Strongly Order region (e.g., Private Peripheral Bus, PPB) are always precise in the Cortex®-M3 and Cortex-M4 processors.

12.2.3 Usage faults

The Usage Fault exception can be caused by a wide range of factors:

- Execution of an undefined instruction (including trying to execute floating point instructions when the floating point unit is disabled).
- Execution of Co-processor instructions – the Cortex®-M3 and Cortex-M4 processors do not support Co-processor access instructions, but it is possible to use the usage fault mechanism to emulate co-processor instruction support.
- Trying to switch to ARM® state – classic ARM processors like ARM7TDMI™ support both ARM instruction and Thumb instruction sets, while Cortex-M processors only support Thumb ISA. Software ported from classic ARM processors might contain code that switches the processor to ARM state, and software could potentially use this feature to test whether the processor it is running on supports ARM code.
- Invalid EXC_RETURN code during exception-return sequence (see section 8.1.4 for details of EXC_RETURN code). For example, trying to return to Thread level with exceptions still active (apart from the current serving exception).
- Unaligned memory access with multiple load or multiple store instructions (including load double and store double; see section 6.6).
- Execution of SVC when the priority level of the SVC is the same or lower than current level.
- Exception return with Interrupt-Continuable Instruction (ICI) bits in the unstacked xPSR, but the instruction being executed after exception return is not a multiple-load/store instruction.

It is also possible, by setting up the Configuration Control Register (CCR; see sections 9.8.4 and 9.8.5) to generate usage faults for the following:

- Divide by zero
- All unaligned memory accesses

Please note that the floating point instructions supported by the Cortex-M4 are not co-processor instructions (e.g., MCR, MRC; see section 5.6.15). However, slightly confusingly, the register that enables the floating point unit is called the Coprocessor Access Control Register (CPACR; see section 9.10).

12.2.4 HardFaults

As illustrated in Figure 12.1, the HardFault exception can be triggered by escalation of configurable fault exceptions. In addition, the HardFault can be triggered by:

- Bus error received during a vector fetch
- Execution of breakpoint instruction (BKPT) with a debugger attached (halt debugging not enabled) and debug monitor exception (see section 14.3) not enabled

Note: In some development tool chains, breakpoints are used by the debugger to carry out semi-hosting. For example, when reaching a "printf" operation, the processor executes a BKPT instruction and halt, and the debugger can detect the halt and

check the register status and the immediate value in the BKPT instruction. Then the debugger can display the message or character form the message in the printf statement. If the debugger is not attached, such operation results in HardFault and executes the HardFault exception handler.

12.3 Enabling fault handlers

By default the configurable fault exceptions are disabled. You can enable these exceptions by writing to System Handler Control and State Register (SCB->SHCSR). Be careful not to change the current status of system exception active status, since this can cause a fault exception.

12.3.1 MemManage fault

You can enable the MemManage Fault exception handler using:

```
SCB->SHCSR |= SCB_SHCSR_MEMFAULTENA_Msk; //Set bit 16
```

The default name for MemManage Fault exception handler (as defined in CMSIS-Core) is:

```
void MemManage_Handler(void);
```

You can set up the priority of the MemManage Fault using:

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

12.3.2 Bus fault

You can enable the Bus Fault exception handler using:

```
SCB->SHCSR |= SCB_SHCSR_BUSFAULTENA_Msk; //Set bit 17
```

The default name for the Bus Fault exception handler (as defined in CMSIS-Core) is:

```
void BusFault_Handler(void);
```

You can set up the priority of the Bus Fault using:

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

12.3.3 Usage fault

You can enable the Usage Fault exception handler using:

```
SCB->SHCSR |= SCB_SHCSR_USGFAULTENA_Msk; //Set bit 18
```

The default name for the Usage Fault exception handler (as defined in CMSIS-Core) is:

```
void UsageFault_Handler(void);
```

You can set up the priority of the Usage Fault using:

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

12.3.4 HardFault

There is no need to enable the HardFault handler. This is always enabled and has a fixed exception priority of −1. The default name for the Hard Fault exception handler (as defined in CMSIS-Core) is:

```
void HardFault_Handler(void);
```

12.4 Fault status registers and fault address registers

12.4.1 Summary

The Cortex®-M3 and Cortex-M4 processors have a number of registers that are used for fault analysis. They can be used by the fault handler code, and in some cases, by the debugger software running on the debug host for displaying fault status. A summary of these registers is shown in Table 12.1. These registers can only be accessed in privileged state.

The Configurable Fault Status Register (CFSR) can be further divided into three parts, as show in Table 12.2. Besides accessing CFSR as a 32-bit word, each part of the CFSR can be accessed using byte and half-word transfers. There is no CMSIS-Core symbol for the divided MMSR, BFSR, and UFSR.

Table 12.1 Registers for Fault Status and Address Information

Address	Register	CMSIS-Core Symbol	Function
0xE000ED28	Configurable Fault Status Register	SCB->CFSR	Status information for Configurable faults
0xE000ED2C	HardFault Status Register	SCB->HFSR	Status for HardFault
0xE000ED30	Debug Fault Status Register	SCB->DFSR	Status for Debug events
0xE000ED34	MemManage Fault Address Register	SCB->MMFAR	If available, showing accessed address that triggered the MemManage fault
0xE000ED38	BusFault Address Register	SCB->BFAR	If available, showing accessed address that triggered the bus fault
0xE000ED3C	Auxiliary Fault Status Register	SCB->AFSR	Device-specific fault status

Table 12.2 Dividing Configurable Fault Status Register (SCB->CFSR) Into Three Parts

Address	Register	Size	Function
0xE000ED28	MemManage Fault Status Register (MMFSR)	Byte	Status information for MemManage Fault
0xE000ED29	Bus Fault Status Register (BFSR)	Byte	Status for Bus Fault
0xE000ED2A	Usage Fault Status Register (UFSR)	Halfword	Status for Usage Fault

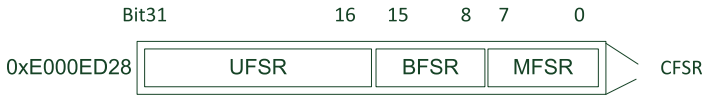

FIGURE 12.2

Configurable Fault Status Register partitioning

12.4.2 Information for MemManage fault

The programmer's model for the MemManage Fault Status Register is shown in Table 12.3.

Each fault indication status bit (not including MMARVALID) is set when the fault occurs, and stays high until a value of 1 is written to the register.

If the MMFSR indicates that the fault is a data access violation (DACCVIOL set to 1) or an instruction access violation (IACCVIOL set to 1), the faulting code can be located by the stacked program counter in the stack frame.

If the MMARVALID bit is set, it is also possible to determine the memory location that caused the fault by using the MemManage Fault Address Register (SCB->MMFAR).

Table 12.3 MemManage Fault Status Register (lowest byte in SCB->CFSR)

Bits	Name	Type	Reset Value	Description
7	MMARVALID	–	0	Indicates the MMFAR is valid
6	–	–	– (read as 0)	Reserved
5	MLSPERR	R/Wc	0	Floating point lazy stacking error (available on Cortex®-M4 with floating point unit only)
4	MSTKERR	R/Wc	0	Stacking error
3	MUNSTKERR	R/Wc	0	Unstacking error
2	–	–	– (read as 0)	Reserved
1	DACCVIOL	R/Wc	0	Data access violation
0	IACCVIOL	R/Wc	0	Instruction access violation

MemManage faults which occur during stacking, unstacking, and lazy stacking (see sections 8.3.6 and 13.3) are indicated by MSTKERR, MUNSTKERR, and MLSPERR, respectively.

12.4.3 Information for bus fault

The programmer's model for the Bus Fault Status Register is shown in Table 12.4. Each fault indication status bit (not including BFARVALID) is set when the fault occurs, and stays high until a value of 1 is written to the register.

The IBUSERR indicates that the bus fault is caused by a bus error during an instruction fetch. Both PRECISERR and IMPRECISERR are for data accesses. PRECISERR indicates a precise bus error (see section 12.3.2), and the faulting instruction can be located from the stacked program counter value. The address of the faulting data access is also written to the Bus Fault Address Register (SCB->BFAR); however, the fault handler should still check if BFARVALID is still 1 after reading BFAR.

If the bus fault is imprecise (IMPRECISERR set to 1), the stacked program counter does not reflect the faulting instruction address, and the address of the faulting transfer will not show in the BFAR.

Bus faults occurring during stacking, unstacking, and lazy stacking (see sections 8.3.6 and 13.3) are indicated by STKERR, UNSTKERR, and LSPERR, respectively.

12.4.4 Information for usage fault

The programmer's model for the Usage Fault Status Register is shown in Table 12.5.

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

Appendix I shows a breakdown of possible reasons for each type of usage fault.

Table 12.4 Bus Fault Status Register (2nd byte in SCB->CFSR)

CFSR Bits	Name	Type	Reset Value	Description
15	BFARVALID	–	0	Indicates BFAR is valid
14	–	–	–	–
13	LSPERR	R/Wc	0	Floating point lazy stacking error (available on Cortex®-M4 with floating point unit only)
12	STKERR	R/Wc	0	Stacking error
11	UNSTKERR	R/Wc	0	Unstacking error
10	IMPRECISERR	R/Wc	0	Imprecise data access error
9	PRECISERR	R/Wc	0	Precise data access error
8	IBUSERR	R/Wc	0	Instruction access error

Table 12.5 Usage Fault Status Register (Upper half-word in SCB->CFSR)

CFSR Bits	Name	Type	Reset Value	Description
25	DIVBYZERO	R/Wc	0	Indicates a divide by zero has taken place (can be set only if DIV_0_TRP is set)
24	UNALIGNED	R/Wc	0	Indicates that an unaligned access fault has taken place
23:20	–	–	–	–
19	NOCP	R/Wc	0	Attempts to execute a coprocessor instruction
18	INVPC	R/Wc	0	Attempts to do an exception with a bad value in the EXC_RETURN number
17	INVSTATE	R/Wc	0	Attempts to switch to an invalid state (e.g., ARM)
16	UNDEFINSTR	R/Wc	0	Attempts to execute an undefined instruction

12.4.5 HardFault status register

The programmer's model for the Usage Fault Status Register is shown in Table 12.6.

HardFault handler can use this register to determine whether a HardFault is caused by any of the configurable faults. If the FORCED bit is set, it indicates that the fault has been escalated from one of the configurable faults and it should check the value of CFSR to determine the cause of the fault.

Similar to other fault status registers, each fault indication status bit is set when the fault occurs, and stays high until a value of 1 is written to the register.

12.4.6 Debug fault status register (DFSR)

Unlike other fault status registers, the DFSR is intended to be used by debug tools such as a debugger software running on a debug host (e.g., a personal computer), or a

Table 12.6 Hard Fault Status Register (0xE000ED2C ,SCB->HFSR)

Bits	Name	Type	Reset Value	Description
31	DEBUGEVT	R/Wc	0	Indicates hard fault is triggered by debug event
30	FORCED	R/Wc	0	Indicates hard fault is taken because of bus fault, memory management fault, or usage fault
29:2	–	–	–	–
1	VECTBL	R/Wc	0	Indicates hard fault is caused by failed vector fetch
0	–	–	–	–

Table 12.7 Debug Fault Status Register (0xE000ED30 ,SCB->DFSR)

Bits	Name	Type	Reset Value	Description
31:5	-	-	-	Reserved
4	EXTERNAL	R/Wc	0	Indicates the debug event is caused by an external signal (the EDBGRQ signal is a input on the processor, typically used in multi-processor design for synchronized debug).
3	VCATCH	R/Wc	0	Indicates the debug event is caused by a vector catch, a programmable feature that allows the processor to halt automatically when entering certain type of system exception including reset.
2	DWTTRAP	R/Wc	0	Indicates the debug event is caused by a watchpoint
1	BKPT	R/Wc	0	Indicates the debug event is caused by a breakpoint
0	HALTED	R/Wc	0	Indicates the processor is halted is by debugger request (including single step)

debug agent software running on the microcontroller to determine what debug event has occurred.

The programmer's model for the Debug Fault Status Register is shown in Table 12.7.

Similar to other fault status registers, each fault indication status bit is set when the fault occurs, and stays high until a value of 1 is written to the register.

12.4.7 Fault address registers MMFAR and BFAR

When a MemManage fault or a bus fault occurs, you might be able to determine the address of the transfer that triggered the fault using MMFAR or BFAR registers.

The programmer's model for the MMFAR Register is shown in Table 12.8.

The programmer's model for the BFAR Register is shown in Table 12.9.

Table 12.8 MemManage Fault Address Register (0xE000ED34, SCB->MMFAR)

Bits	Name	Type	Reset Value	Description
31:0	ADDRESS	R/W	Unpredictable	When the value of MMARVALID is 1, this field holds the address of the address location that generates the MemManage fault.

Table 12.9 Bus Fault Address Register (0xE000ED38, SCB->BFAR)

Bits	Name	Type	Reset Value	Description
31:0	ADDRESS	R/W	Unpredictable	When the value of BFARVALID is 1, this field holds the address of the address location that generates the Bus Fault.

Inside the Cortex®-M3 and Cortex-M4 processors, the MMFAR and BFAR shared the same physical hardware. This reduces the silicon size of the processor. Therefore only one of the MMARVALID or BFARVALID can be 1 at a time. As a result, if one of the fault exceptions is pre-empted by another due to a new fault exception, the value in the MMFAR or BFAR could have become invalid. To ensure that the fault handlers are getting the accurate fault address information, it should:

1. First read the value of MMFAR (for MemManage fault), or BFAR (for bus fault), then
2. Read the value of MMFSR (for MemManage fault), or BFSR (for bus fault), to see if MMARVALID or BFARVALID is still 1. If they are still 1, then the fault address is valid.

Note that if an unaligned access faults, the address in the MMFAR is the actual address that faulted. The transfer is divided into a number of aligned transfers by the processor, and the MMFAR can be any value in the address range of these aligned transfers. For bus fault with BFARVALID set, the BFAR indicates the address requested by the instruction, but can be different from the actual faulting address. For example, in a system with a valid 64KB SRAM address 0x20000000 to 0x2000FFFF, a word-size access to 0x2000FFFE might fault in the second half-word at address 0x20010000. In this case, BFAR showing 0x2000FFFE is still in the valid address range.

12.4.8 Auxiliary fault status register

The AFSR was added from Cortex®-M3 r2p0 onwards. It allows silicon designers to add their own fault status information. The programmer's model for the AFSR Register is shown in Table 12.10.

Similar to other fault status registers, each fault indication status bit is set when the fault occurs, and stays high until a value of 1 is written to the register.

Table 12.10 Auxiliary Fault Status Register (0xE000ED3C, SCB->AFSR)

Bits	Name	Type	Reset Value	Description
31:0	Implementation Defined	R/W	0	Implementation defined fault status

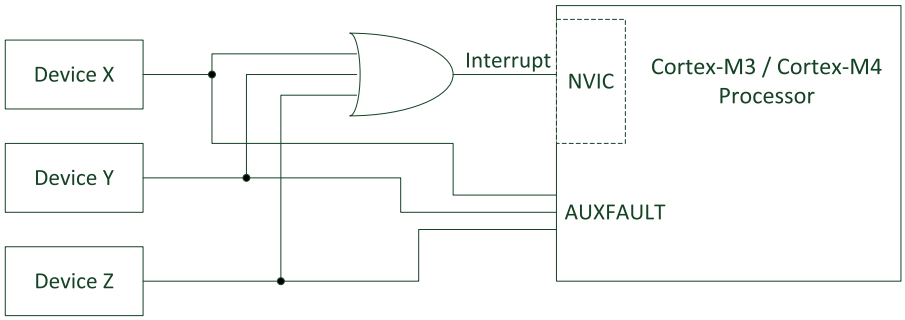

FIGURE 12.3

Additional fault generation sources connected to the processor via AUXFAULT and interrupt

On the processor interface, a 32-bit input (AUXFAULT) is available for silicon designers to connect to various devices that can be used to generate fault events, as shown in Figure 12.3.

When a fault event happens in one of these devices, it triggers an interrupt, and the interrupt handler can use the AFSR to determine which device generated the fault. Since this is not intended to be used for general interrupt processing, using the software to determine the cause of fault does not cause a latency issue.

12.5 Analyzing faults

It is not uncommon to encounter fault exceptions during software development. In some cases it can be a bit of challenge to find out what went wrong. In most cases, the information provided by the fault status registers and fault address registers is certainly very useful. In addition, more information can be obtained using various techniques and tools including:

- Stack Trace: After a fault exception is triggered, we can halt the processor and examine the processor status and the memory contents either by using breakpoint hardware, or manually inserting a breakpoint instruction. Besides the current register values, we can trace the stacked register values including the stacked Program Counter (PC) from the stack pointers. Combining the stacked PC and the fault status registers values can very often lead you to the right answers fairly quickly.
- Event Trace: The data trace feature in the Cortex®-M3 and Cortex-M4 processors allows you to collect exception history using low-cost debuggers. The exception trace can be output through the single pin Serial Wire Output pin (see Chapter 14). If a program failure is related to exception handling, the event trace feature allows you to see which exceptions occurred before the failure and hence make it easier to locate the issue.

- Instruction Trace: Use the Embedded Trace Macrocell (ETM) to collect information about instruction executed, and display it on a debugger to identify the processor operations before the failure. This requires a debugger with Trace Port capture function.

In typical stack trace operations, we can add a breakpoint to the beginning of the HardFault handler (or other configurable handlers if they are used). When a fault occurs, the processor enters the fault handler and halts.

First, we need to determine which stack pointer was being used when the fault occurred. In the majority of the applications without an OS, only the Main Stack Pointer (MSP) would be used. However, if the application uses PSP, we need to determine the SP used by checking bit 2 of the Link Register (LR), as shown in Figure 12.4.

From the stack pointer value, we can easily locate the stacked registers like stacked PC (return address) and stacked xPSR:

- In many cases the stacked PC provides the most important hint for debugging the fault. By generating a disassembled code listing of the program image in the toolchain, you can easily pinpoint the code fragment where the fault occurred, and understand the failure from the information provided in the fault status registers, and the current and stacked register values.
- Stacked xPSR can be useful for identifying if the processor was in handler mode when the fault occurred, and whether there has been an attempt to switch the processor into ARM® state (if the T-bit in the EPSR is cleared, there has been an attempt to switch the processor into ARM state).

Finally, the LR value when entering the fault handler might also provide hints about the cause of the fault. In the case of faults caused by invalid EXC_RETURN values, the value of LR when the fault handler is entered shows the previous LR value when the fault occurred. The fault handler can report the faulty LR value, and software programmers can then use this information to check why the LR ends up with an illegal return value.

In some debug tools, the debugger software contains features which allow you to access fault status information easily. For example, in Keil™ MDK-ARM, you can access to the fault status registers using the "Fault Report" window, as shown in Figure 12.5. This can be accessed from the pull-down menu "Peripherals"-> "Core Peripherals" -> "Fault Reports."

Various trace features in the debugger can also help to identify the source of the problem in your application code. More information on this is covered in Chapter 14. There is also an application note on the Keil website about debugging fault exceptions: Application Note 209 "Using Cortex-M3 and Cortex-M4 Fault Exceptions" (http://www.keil.com/appnotes/docs/apnt_209.asp).

Other debug tools also have the debug feature to assist fault analysis. For example, the debugger in the Atollic TrueStudio has a Fault Analyzer feature; it extracts information from the processor such as the fault status registers to identify the reasons that caused the fault.

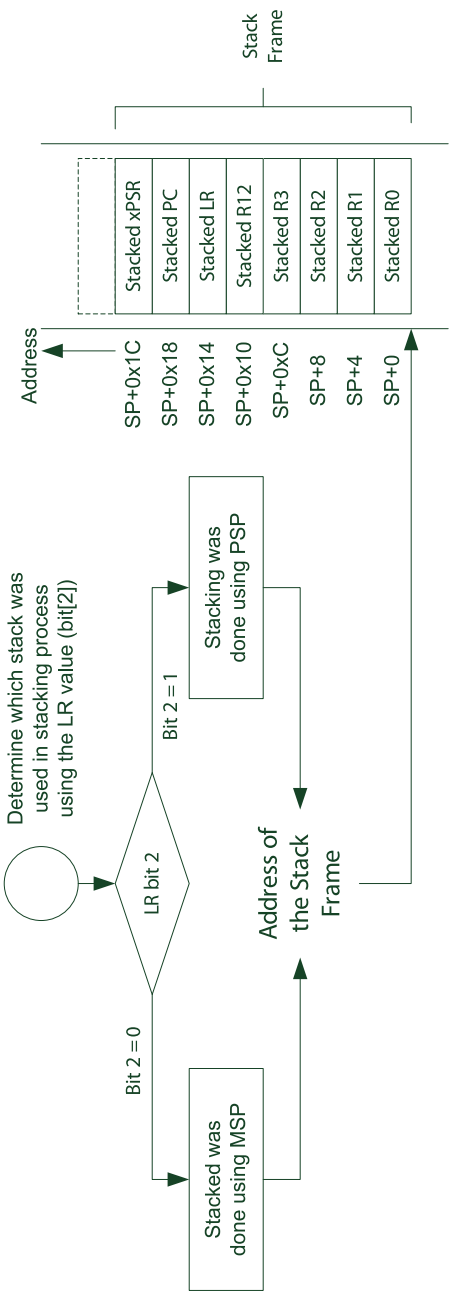

FIGURE 12.4

Stack trace flow to locate stack frame and stacked registers

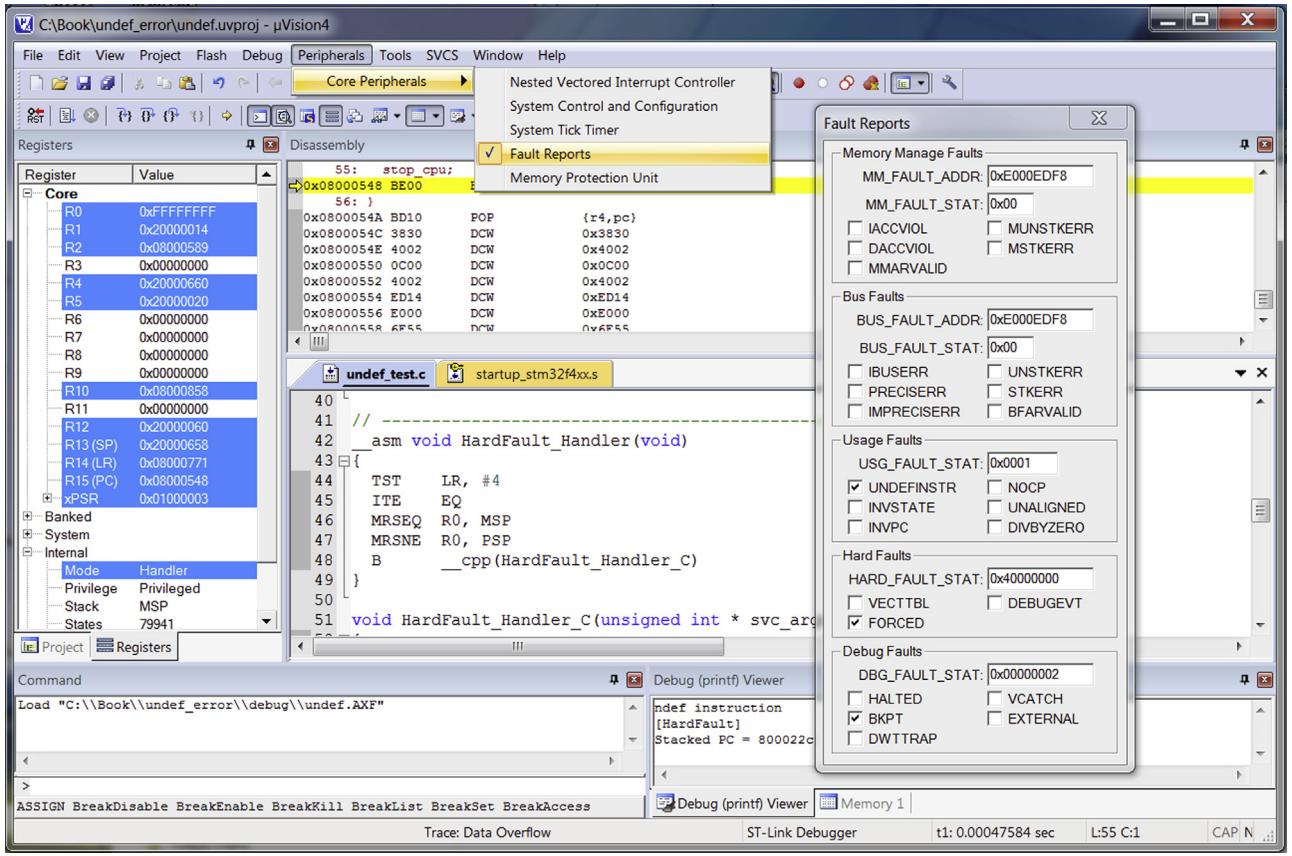

FIGURE 12.5

Fault Reports window in Keil MDK-ARM showing the fault status registers

12.6 Faults related to exception handling

In some cases, faults can be generated during exception handling. The most common case is incorrect stack setup; for example, the stack space reserved is too small and causes the stack space to run out. In this section we will look at what could have gone wrong and what fault exception could be triggered.

12.6.1 Stacking

During exception entry, a number of registers are pushed to the stack. Potentially this can trigger a Bus Fault if the memory system returns an error response, or a MemManage fault if the MPU is programmed and the stack grows beyond the allocated memory space for the stack.

If a bus error is received, the STKERR bit (bit 4) in the Bus Fault Status Register (BFSR) is set. If an MPU violation is detected, the error is indicated by the MSTKERR bit (bit 4) in the MemManage fault status register.

12.6.2 Unstacking

During exception exits, the processor restores register values by reading back values from the stack frame. It is possible to trigger a Bus Fault if the memory system returned a bus error response, or a MemManage fault if the MPU detected an access violation.

If a bus error is received, the UNSTKERR bit (bit 3) in the BFSR is set. If an MPU violation is detected, the error is indicated by the MUNSTKERR bit (bit 3) in the MemManage fault status register.

It is uncommon to get an unstacking error without getting a stacking error. If the Stack Pointer (SP) value was incorrect, in most cases the fault would have happened during stacking. However, it is not impossible to get an unstacking fault without a stacking fault. For example, this can happen if:

- The value of SP was changed during the execution of the exception handler
- The MPU configuration was changed during the execution of the exception handler
- The value of EXC_RETURN was changed during the execution of the exception handler, so the SP being used in unstacking was different from the one used in stacking

12.6.3 Lazy stacking

For the Cortex®-M4 processor with floating point unit, Bus Fault and MemManage Fault could be triggered during lazy stacking. The lazy stacking feature allows the stacking of floating point registers to be deferred, and only push those registers to the allocated space if the exception handler uses the floating point unit. When this happens, the processor pipeline is stalled and carries out the stacking, and then executes the floating point instruction after the stacking is completed.

If a bus error is received during the lazy stacking operation, the Bus Fault exception is triggered and the error is indicated by LSPERR (bit 5) of the Bus Fault Status Register. If a MPU access violation occurs, the MemManage Fault exception is triggered and the error is indicated by MLSPERR (bit 5) of the MemManage Fault Status Register.

12.6.4 Vector fetches

If a bus error takes place during a vector fetch, the HardFault exception will be triggered, and the error will be indicated by the VECTTBL (bit 1) of the Hard Fault Status Register. The MPU always permits vector fetches and therefore there is no MPU access violation for vector fetches. If a vector fetch error occurs, one thing that needs checking is the value of the VTOR to see whether the vector table has been relocated to the correct address range.

If the LSB of the exception vector is 0, it indicates an attempt to switch the processor to the ARM® state (use ARM instructions instead of Thumb instructions), and this is not supported in Cortex®-M processors. When this happens, the processor will trigger a Usage Fault at the first instruction of the exception handler, with INVSTATE bit (bit 1) if the Usage Fault Status Register is set to 1 to indicate the error.

12.6.5 Invalid returns

If the EXC_RETURN value is invalid or does not match the state of the processor (as in using 0xFFFFFFF1 to return to Thread mode), it will trigger a Usage Fault. The bits INVPC (bit 2) or INVSTATE (bit 1) of the Usage Fault Status Register will be set, depending on the actual cause of the fault.

12.6.6 Priority levels and stacking or unstacking faults

Configurable fault handlers have programmable priority levels. If a fault happens and the current priority level is the same or higher than the associated configurable fault handler, the fault event is escalated to the HardFault exception, which has a fixed priority of −1.

If a stacking or unstacking error occurs during an exception sequence, the current priority level is based on the priority level of the interrupted process/task, as shown in Figure 12.6.

If the Bus Fault or MemManage Fault exception has the same or lower priority than the current priority level, the HardFault exception will be executed first.

If the Bus Fault or MemManage Fault exception is enabled and has higher priority than both the current level and the priority level of the exception to be serviced, then the Bus Fault or MemManage Fault exception would be executed first.

If the Bus Fault or MemManage Fault exception is enabled and has a priority level between the current level and the exception to be serviced, the handler for

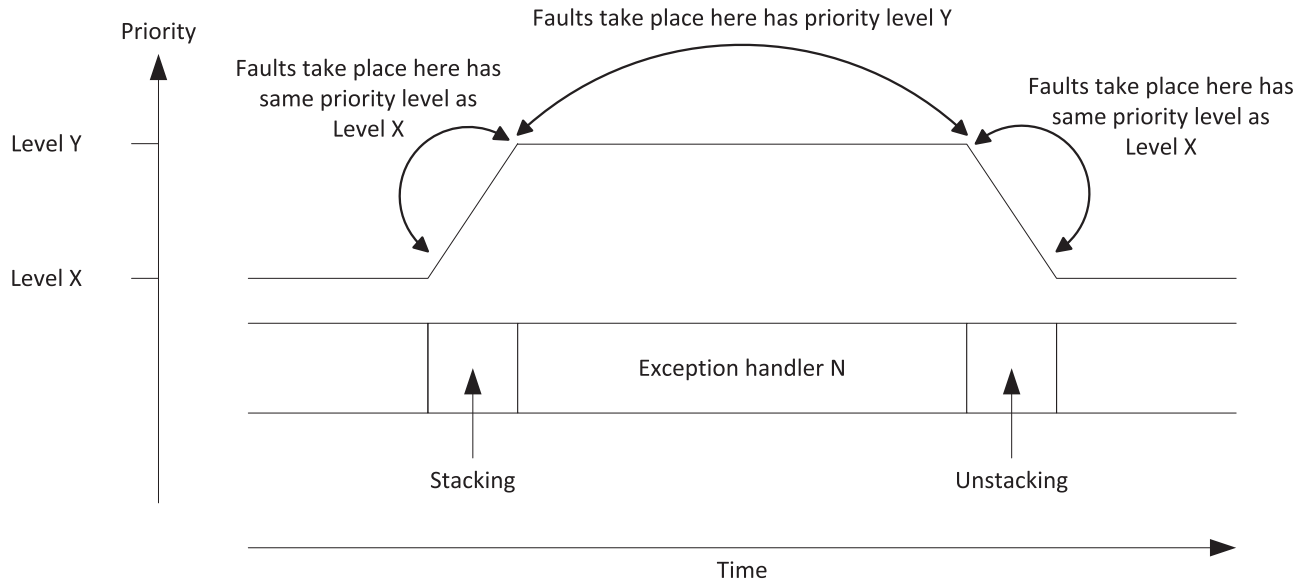

FIGURE 12.6

Priority level at stacking and unstacking

the exception to be serviced is executed first, and the Bus Fault or MemMange Fault handler is executed afterwards.

12.7 Lockup

12.7.1 What is lockup?

When an error condition occurs, one of the fault handlers will be triggered. If another fault happens inside a configurable fault handler, then either another configurable fault handler is triggered (if the fault is a different kind and the other fault handler and has higher priority than the current level), or the Hard Fault handler is triggered and executed. However, what happens if another fault happens during the execution of the HardFault handler? (This is a very unlucky situation, but it can happen.) In this case, a lockup will take place.

Lockup can happen if:

- A fault occurs during execution of the HardFault or NMI (Non-Maskable Interrupt) exception handler
- A bus error occurs during vector fetch for HardFault or NMI exceptions
- Trying to execute SVC instruction in the HardFault or NMI exception handler
- Vector fetch at startup sequence

During lockup, the processor stops program execution and asserts an output signal called LOCKUP. How this signal is used depends on the microcontroller design; in some cases it can be used to generate a system reset automatically. If the lockup is caused by an error response from the bus system, the processor might retry the access continuously, or if the fault is unrecoverable it could force the program counter to 0xFFFFFFFX and might keep fetching from there.

If the lockup is caused by a fault event inside the HardFault handler (double fault condition), the priority level of the processor is still at priority level -1, and it is still possible for the processor to respond to an NMI (priority level -2) and execute the NMI handler. But after the NMI handler finishes, it will return to the lockup state and the priority level will return to -1.

There are various ways to exit the lockup state:

- System reset or power on reset
- The debugger can halt the processor and clear the errors (e.g., using reset or clearing current exception handling status, update program counter value to a new starting point, etc.)

Typically a system reset is the best method as it ensures that the peripherals and all interrupt handling logic returns to the reset state.

You might wonder why we do not simply reset the processor when a lockup takes place. It might be good for a live system, but during software development, we should first try to find the cause of the problem. If we reset the system automatically, it will be impossible to analyze what went wrong because the hardware status will change.

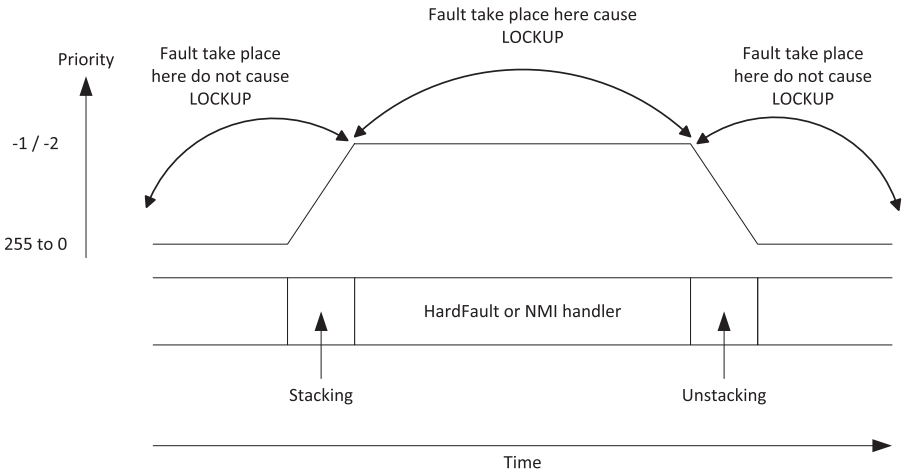

FIGURE 12.7

Only a Fault occurring during a HardFault or NMI handler will cause Lockup

The Cortex®-M processor designs export the lockup status to its interface and chip designers can implement a programmable auto reset feature, so that when this auto reset feature is enabled, the system can reset itself automatically.

Note that a bus error or MPU access violation occurs during stacking or unstacking (except vector fetch) when entering a HardFault handler or NMI handler does not cause the system to enter lockup state (see Figure 12.7). However, the Bus Fault exception could end up in a pending state and execute after the HardFault handler.

12.7.2 Avoiding lockup

In some applications, it is important to avoid lockup, and extra care is needed when developing the HardFault handler and NMI handler. For example, we might need to avoid stack memory access unless we know that the stack pointer is still in a valid memory range. For example, if the starting of the HardFault handler has a stack push operation and the MSP (Main Stack Pointer) was corrupted and pointed to invalid memory location, we could end up entering lockup state immediately at the start of the HardFault handler:

```
HardFault_Handler
     PUSH {R4-R7,LR}  ; Bad idea unless you are sure that the
                      ; stack is safe to use!
     . . .
```

Even if the stack pointer is in a valid memory range, we might still need to reduce the amount of stack used by the HardFault and NMI handler, if the available stack size is small.

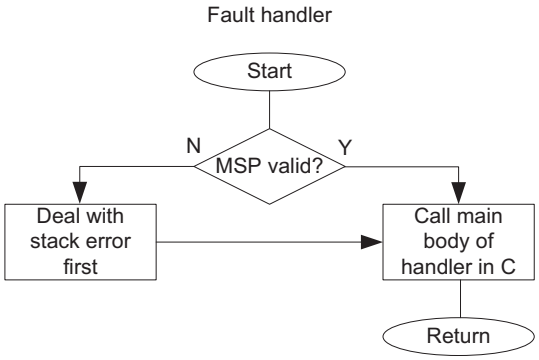

FIGURE 12.8

Adding a SP value check in fault handlers

In safety critical systems, we can add an assembly wrapper code for fault handlers (Figure 12.8) to check if the value of MSP is still in a valid range before calling the fault handlers in C code, which might have stack operations inserted by the C compilers.

One approach for developing HardFault and NMI handlers is to carry out only the essential tasks inside the handlers, and the rest of the tasks, such as error reporting, can be pended using a separate exception such as PendSV. This helps to ensure that the HardFault handler or NMI is small and robust.

Furthermore, we should ensure that the NMI and HardFault handler code will not try to use the SVC instruction. Since SVC always has lower priority than HardFault and NMI, using SVC in these handlers will cause lockup. This might look simple, but when your application is complex and you call functions from different files in your NMI and HardFault handlers, you might accidentally call a function that contains an SVC instruction. Therefore before you develop your software with SVC, you need to plan the SVC service implementation carefully.

12.8 Fault handlers

12.8.1 HardFault handler for debug purposes

A number of activities can be carried out in fault handlers, for example:

- Shutdown the system safely
- Report errors
- Self-reset
- Carry out remedial actions (if possible)
- For an OS environment, the task that triggered the fault can be terminated and restarted.

- Optionally clear the fault status in the fault status registers. This should be included in the fault handler if it carries out remedial action and resumes normal operations.

The implementations for most of these tasks are dependent on the application. The self-reset operation is discussed in section 9.6. Here, we will have a look at reporting information about the fault. One of the common ways to do this is to create a HardFault handler that:

- Reports that the HardFault happened
- Reports the Fault Status Register and Fault Address Register values
- Reports additional information from the stack frame

The following example HardFault handler assumes that you have some way to display messages generated from "printf" in C (see Chapter 18 for details). As mentioned in section 12.5, we will need to check the value of EXC_RETURN code to determine if MSP or PSP was used for the stacking.

In order to extract the value of EXC_RETURN from LR and to locate the starting of stack frame from MSP/PSP, we need a small assembly code wrapper. This extracts the starting address of the stack frame and passes it to the second part of the HardFault handler, which is programmed in C. This wrapper also passes the EXC_RETURN value as the second parameter:

```
/* Assembly wrapper for Keil™ MDK, ARM® Compilation tool chain
(including DS-5™ Professional and RealView® Development Suite) */
// ------------------------------------------------------------------
  ---------
// Hard Fault handler wrapper in assembly
// It extracts the location of stack frame and passes it to handler
// in C as a pointer. We also extract the LR value as second
// parameter.
__asm void HardFault_Handler(void)
{
      TST      LR, #4
      ITE      EQ
      MRSEQ    R0, MSP
      MRSNE    R0, PSP
      MOV      R1, LR
      B        __cpp(HardFault_Handler_C)
}
```

For users of gcc, you can create a separate assembly file to do the same thing:

```
/* Assembly file for gcc */
 .text
 .syntax unified
 .thumb
 .type   HardFault_Handler, %function
```

```
 .global HardFault_Handler
 .global HardFault_Handler_c

HardFault_Handler:
 tst    lr, #4
 ite    eq
 mrseq  r0, msp /* stacking was using MSP */
 mrseq  r0, psp /* stacking was using PSP */
 mov    r1, lr /* second parameter */
 ldr    r2,=HardFault_Handler_c
 bx     r2
 .end
```

And for users of IAR Embedded Workbench (many thanks for various Cortex®-M users for porting the example to IAR and posting it to the Internet[1]):

```
// Assembly wrapper for IAR Embedded Workbench
// ------------------------------------------------------------
  ---------
// Hard Fault handler wrapper in assembly
// It extracts the location of stack frame and passes it to handler
// in C as a pointer. We also extract the LR value as second
// parameter.
void HardFault_Handler(void)
{
__asm("TST LR, #4");
__ASM("ITE EQ");
__ASM("MRSEQ R0, MSP");
__ASM("MRSNE R0, PSP");
__ASM("MOV R1, LR");
__ASM("B HardFault_Handler_C");
}
```

The second part of the HardFault handler is coded in C; it displays the fault status registers, fault address register, and the contents in the stack frame:

```
// Second part of the HardFault handler in C
void HardFault_Handler_C(unsigned long * hardfault_args, unsigned
int lr_value)
 {
  unsigned long stacked_r0;
  unsigned long stacked_r1;
  unsigned long stacked_r2;
  unsigned long stacked_r3;
```

[1]See http://blog.frankvh.com/2011/12/07/cortex-m3-m4-hard-fault-handler/

```
unsigned long stacked_r12;
unsigned long stacked_lr;
unsigned long stacked_pc;
unsigned long stacked_psr;
unsigned long cfsr;
unsigned long bus_fault_address;
unsigned long memmanage_fault_address;

bus_fault_address       = SCB->BFAR;
memmanage_fault_address = SCB->MMFAR;
cfsr                    = SCB->CFSR;

stacked_r0 = ((unsigned long) hardfault_args[0]);
stacked_r1 = ((unsigned long) hardfault_args[1]);
stacked_r2 = ((unsigned long) hardfault_args[2]);
stacked_r3 = ((unsigned long) hardfault_args[3]);
stacked_r12 = ((unsigned long) hardfault_args[4]);
stacked_lr = ((unsigned long) hardfault_args[5]);
stacked_pc = ((unsigned long) hardfault_args[6]);
stacked_psr = ((unsigned long) hardfault_args[7]);

printf ("[HardFault]\n");
printf ("- Stack frame:\n");
printf (" R0 = %x\n", stacked_r0);
printf (" R1 = %x\n", stacked_r1);
printf (" R2 = %x\n", stacked_r2);
printf (" R3 = %x\n", stacked_r3);
printf (" R12 = %x\n", stacked_r12);
printf (" LR = %x\n", stacked_lr);
printf (" PC = %x\n", stacked_pc);
printf (" PSR = %x\n", stacked_psr);
printf ("- FSR/FAR:\n");
printf (" CFSR = %x\n", cfsr);
printf (" HFSR = %x\n", SCB->HFSR);
printf (" DFSR = %x\n", SCB->DFSR);
printf (" AFSR = %x\n", SCB->AFSR);
if (cfsr & 0x0080) printf (" MMFAR = %x\n",
 memmanage_fault_address);
if (cfsr & 0x8000) printf (" BFAR = %x\n", bus_fault_address);
printf ("- Misc\n");
printf (" LR/EXC_RETURN= %x\n", lr_value);

while(1); // endless loop
}
```

Please note that this handler will not work correctly if the stack pointer is pointing to an invalid memory region (e.g., because of stack overflow). This affects all C code, as most C functions need stack memory. To help debug the issue, we can also generate a disassembled code list file so that we can locate the problem used to report the stacked program counter value.

The values of BFAR and MMFAR stay unchanged if the BFARVALID or MMARVALID is set. However, if a new fault occurs during the execution of this fault handler, the value of the BFAR and MMFAR could potentially be erased. In order to ensure the fault addresses accessed are valid, the following procedure should be used:

1. Read BFAR/MMFAR.
2. Read CFSR to get BFARVALID or MMARVALID. If the value is 0, the value of BFAR or MMFAR accessed can be invalid and can be discarded.
3. Optionally clear BFARVALID or MMARVALID.

Otherwise, it is possible to get an incorrect fault address if the following sequence occurs:

1. Read BFARVALID/MMARVALID,
2. Detected that valid bit is set, then going to read BFAR or MMFAR,
3. Just before reading BFAR or MMFAR, a higher-priority handler arrives at the current fault handler, and the higher-priority exception handler generates another fault,
4. Another fault handler is triggered, and this clears the BFARVALID or MMARVALID. This means that the value in BFAR and MMFAR will not be held constant, and will be lost.
5. After returning to the original fault handler, the value of BFAR or MMFAR is read, but now the value is invalid and leads to incorrect information in the fault report.

Therefore, it is important to read the BFAR or MMFAR first, and then read BFARVALID and MMARVALID in CFSR.

12.8.2 Fault mask

In a configurable fault handler, if needed we can set the FAULTMASK to:

- Disable all interrupts, thus allowing the processor to carry out remedial actions without getting interrupted (please note that the processor can still be interrupted by a NMI exception).
- Disable/Enable the configurable fault handler to bypass MPU, and to ignore bus faults (see BFHFNMIGN bit of Configuration Control Register in section 9.8.3)

These characteristics allow a configurable fault handler to try to access certain memory locations that may or may not be valid.

Potentially the FAULTMASK can also be used outside fault handlers. For example, if you have a piece of software that needs to run on a number of microcontrollers with various SRAM sizes, you can use the FAULTMASK to disable bus

faults, and then carry out a RAM read-write test to detect the available RAM size during run-time.

12.9 Additional information

12.9.1 Running a system with two stacks

In Chapter 10 we covered the shadow stack point feature, which is useful for an OS. For systems without an embedded OS, the two-stack arrangement can have another usage: the separation of stacks used by Thread mode and Handler mode can help debugging stack issues in some cases, and allows exception handlers (including the fault handlers) to run normally even if the stack pointer for the Thread mode is corrupted and points to invalid memory locations. In safety critical systems this can be important.

To do this, we need to get the Thread mode code to switch from using the MSP (Main Stack Pointer) to using the PSP (Process Stack Pointer). It is relatively straightforward to do this in the reset handler. For example, if you are using Keil™ MDK-ARM, you can add code in the startup code to reserve an extra handler mode stack memory, and set the MSP, PSP, and CONTROL registers accordingly in the reset handler. You might also need to update the `__user_initial_stackheap` function at the end of the startup code.

```
; Modification to a startup code file (for Keil MDK-ARM) to switch
Thread mode to use PSP
 Handler_Stack_Size  EQU  0x00000200
 Thread_Stack_Size   EQU  0x00000400

                AREA STACK, NOINIT, READWRITE, ALIGN=3
 Handler_Stack_Mem    SPACE Handler_Stack_Size
 __initial_handler_sp
 Thread_Stack_Mem     SPACE Thread_Stack_Size
 __initial_sp
 ...
 ; Reset handler
 Reset_Handler PROC
               EXPORT Reset_Handler    [WEAK]
               IMPORT SystemInit
               IMPORT __main
               LDR    R0, =__initial_sp
               MSR    PSP, R0
               LDR    R0, =__initial_handler_sp
               MSR    MSP, R0
               MOVS   R0, #2      ; Set SPSEL bit
               MSR    CONTROL, R0 ; Now Thread mode use PSP
               ISB
```

```
                LDR     R0, =SystemInit
                BLX     R0
                LDR     R0, =__main
                BX      R0
                ENDP
...
__user_initial_stackheap

     LDR  R0, = Heap_Mem
     LDR  R1, =(Thread_Stack_Mem + Thread_Stack_Size)
     LDR  R2, = (Heap_Mem + Heap_Size)
     LDR  R3, = Thread_Stack_Mem
     BX   LR
```

It is also possible to do this in C, but switching the stack pointer would be slightly more complex because it can be a bad idea to change the current stack point value after the C program started, as the stack might hold local variables that are already initialized and will be used later. To solve this problem, we need to change the PSP to where the current stack is (i.e., MSP current value), switch the SPSEL bit in the CONTROL register, then move the MSP to a memory space reserved for handler stack. For example, you can declare a memory space for the handler stack as static memory array.

```
Example C code to enable Thread mode to use the Process Stack with PSP
   uint64_t Handler_Stack[128]; // Handler Stack = 128x8 = 1024 bytes
   int main(void) {
   uint32_t tmp;
   ...
    tmp=(uint32_t)(Handler_Stack)+(sizeof Handler_Stack); // Get top of stack
     __set_PSP(__get_MSP());              // Set PSP to be the same as MSP
     __set_CONTROL(__get_CONTROL()|0x2); // Set SPSEL, Do not change
                                             other bits
     __ISB();                            // ISB after CONTROL change
                                         // (architectural recommendation)
     __set_MSP(tmp);                      // Move MSP to point to Handler
                                             stack
   ...
```

For Cortex®-M4 with floating point unit, since the floating point unit might have been activated and used, the bit 2 of the CONTROL register could already have been set. Therefore when setting SPSEL bit in the CONTROL register we need to perform a read-modify-write sequence to prevent clearing the FPCA bit accidentally.

12.9.2 Detect stack overflow

One of the common causes for software failure is stack overflow. To prevent this, traditionally, it is common for software developers to fill the SRAM with a

predefined pattern (e.g., 0xDEADBEEF), then execute the program for a while, stop the target, and see how much stack has been used. This method works to an extent, but might not be accurate because the conditions for maximum stack usage might not have been triggered.

In some tool chains, you can get an estimation of the required stack size from report files after project compilation. For example, if you are using:

- Keil™ MDK-ARM®, after compilation you can find an HTML file in the project directory. One of the pieces of information provided in this file is the maximum stack size which the functions use.
- IAR Embedded Workbench, you need to enable two project options: the "Generate linker map file" option in the "List" tab for Linker, and the "Enable stack usage analysis" option in "Advanced" tab for Linker. After the compilation process, you can then see a "Stack Usage" section in the linker report (.map) in the "Debug\List" subdirectory.

Some software analysis tools can also give you a report on stack usages and a lot more information to help you improve the quality of the program code. However, if there is a stack issue such as stack leak in the software, the compilation report file cannot help you. So we need some ways to detect stack usage.

One method is to locate the stack near to the bottom of the SRAM space. When the stack is fully used, the processor gets a bus error in the next stack push because the transfer is no longer in a valid memory region, so the fault handler is executed. If the fault handler is not using the two-stacks arrangement, we need to reset the stack point to a valid memory location in the beginning of the fault handler, so that the remaining parts of the fault handler can run correctly.

Another method is to use the MPU to define a small, inaccessible or read-only memory region at the end of the stack space. If the stack overflows, the MemManage fault exception is triggered and the MPU can be turned off temporarily to allow additional stack space for the fault handler to execute.

If the system is connected to a debugger, you could set a data watch point (a debug feature) at the end of the stack memory so that the processor halts when all the stack space is used. For a standalone test environment, the data watchpoint feature can also potentially be used to trigger a debug monitor exception if no debugger is connected (if a debugger is connected, the debugger might overwrite the data watchpoint setting programmed by the application code).

For applications with an OS, the OS kernel can also carry out checking of the PSP value during each context switching to ensure that the application tasks only used the allocated stack space. While this is not as reliable as using the MPU, it is still a useful method and is easy to implement in many RTOS designs.