Troubleshooting Guide

I.1 Overview

There can be various different symptoms when an application does not work. For example:

- The program does not run at all, or the processor failed to start.
- The program stopped responding.
- The program entered a fault handler, or entered lockup state.

There can be many different reasons for program failures. Some of them could be down to hardware issues (e.g., power supply or clock), and in some cases caused by incorrect software code implementations. Since the hardware requirements of different microcontrollers are different (e.g., voltage, clock generation circuits), the hardware design issues cannot be covered in this book. In this section, we will look at common errors and possible reasons in the software for the system to fail, and some of the debugging techniques.

I.2 The debugger cannot connect to the development board

If you find that the debugger software is unable to connect to the target board, this is more likely to be hardware problems. Please check:

- Supply voltage for the target board.
- Connection for the debug signals.
- Options for the debug adaptor (is the right debug protocol is select? If the debug clock frequency too high?)
- Did you select the right device type/part number in your project?
- You might also need to check if the crystal oscillator is running, and if the capacitor values for the crystal oscillator are correct.

Ground noises, or in general, noises in power supply can be another issue that can cause problems to debug connection. And in some cases, although the microcontrollers might allow you to operate with less than 3 volts in the power supply, the debug adaptor might not be able to handle low voltage debug interface signals. So you might need to increase the supply voltage to 3.3 volts or higher, or in some cases adjust the settings of the debug adaptor in the debugger IDE windows.

The length of the debug interface cable could be another issue. If you are using a separated USB debug adaptor and use a flat cable to connect the Serial Wire or JTAG

debug interface signals, you might need to make sure the cable is not too long and select a suitable clock speed for the interface (typically in the debug adaptor option window of the IDE).

I.3 The system does not start

In some cases, some users find that the debugger can connect to the target system and can download the program image, but the application does not start at all.

I.3.1 Vector table missing or vector table in wrong place

The Cortex®-M processor needs a vector table to boot up. Usually, you project should include a startup code from the microcontroller vendors and it should contains the vector table. The vector table should be placed in the beginning of the memory. If you do have a vector table in the project, make sure it is a vector table which is suitable for Cortex-M3 or Cortex-M4 (e.g., vector table code for ARM7TDMI™ cannot be used). It is also possible for the vector table to be removed during the link stage, or being placed into the wrong address location.

You should generate a disassembled listing of the compiled image or a linker report to see if the vector table is present, and if it is correctly placed at the start of the memory.

I.3.2 Incorrect C startup code being used

Besides compiler options, make sure you are specifying the correct linker options as well. Otherwise a linker might pull in incorrect C startup code. For example, it might end up using start up code for another ARM® processor, which contains instructions not supported by the Cortex®-M3 or Cortex-M4, or it could use startup code for a debug environment with semi-hosting, which might contains a Breakpoint Instruction (BKPT) or Supervisor Call (SVC), which can cause an unexpected HardFault or software exception.

I.3.3 Incorrect value in reset vector

Make sure the reset vector is really pointing to the intended reset handler. Also, you should check that the exception vectors in the vector table have the LSB set to 1 to indicate Thumb code.

I.3.4 Program image not programmed in flash correctly

Most flash programming tools automatically verify the flash memory after programming. If not, after the program image is programmed into the flash, you might need to double check if the flash memory has been updated correctly. In some cases, you might need to erase the flash first, and then program the program image.

I.3.5 Incorrect toolchain configurations

Some other toolchain configurations can also cause problems with the startup sequence. For example, memory map settings, CPU options, endianness settings, and the like.

I.3.6 Incorrect Stack Pointer initialization value

This involves two parts. Firstly, the initial stack pointer value (the first word on the vector table) needs to point to a valid memory address. Secondly, the C startup code might have a separate stack setup step. Try getting the processor to halt at the startup sequence, and single step through it to make sure the stack pointer is not changed so that it points to an invalid address value.

I.3.7 Incorrect endian setting

Most ARM-based microcontrollers are using little endian, but there is a chance that someday you could be using an ARM® Cortex®-M3/M4 microcontroller in big endian configuration. If this is the case, make sure the C compiler options, assembler options and linker options are set up correctly to support big endian mode.

I.4 System not responding

The most likely case is that the processor has trigger a fault exception, and stay in a loop inside the fault exception handler. By default, the fault exception handlers are defined as dead loops inside the startup code. For example, the default HardFault handler can be something like:

```
HardFault_Handler\
    PROC
    EXPORT HardFault_Handler   [WEAK]
    B  .
    ENDP
```

Since the processor is executing a dead loop inside the HardFault handler (only NMI has higher priority than HardFault), the program execution is likely to stay there forever unless you reset the system or halt the processor.

Another possible reason is that the system is in sleep mode. There are several cases where this could cause the system to become unresponsive:

- The sleep mode was entered when the processor is in high priority exception handlers, or an exception masking register (e.g., PRIMASK) was set incorrectly. As a result, other exceptions are blocked.
- The sleep mode used ends up turning off the peripherals that you want to use to wake up the system. Some of the sleep modes might turn off a number of peripherals, including buffer at input pins, by default.
- One possible mistake is enabling the Sleep-on-Exit feature too early during initialization. If this is set early during system initialization, and an exception

occurred, the system will enter sleep mode when the exception handler completed, even before the rest of the initialization task is finished.

One more commonly issue is error in clock or Phase-Lock-Loop (PLL) setup. A mistake in PLL setup could end up overclocking the system, or unable to lock the clock to a stable frequency. These could end up with unpredictable behaviors.

I.5 Fault exceptions

One of the challenges of using the Cortex®-M3/M4 processor is to locate problems when the program goes wrong. On the bright side, the hardware is working, and various features and techniques are available to help debugging.

First, let's look at what we can do with fault exceptions:

- Halt the processor when fault exception occurs. Some debuggers support the vector catch feature which can be enabled and halt the processor automatically when fault exception occurs. If this is not supported, we can insert a Breakpoint Instruction (BKPT) to the fault handler to halt the processor. After the processor is halted, we can then use various debugger features to identify the cause of the fault.
- Implement a fault exception handler as outlined in Chapter 12. This can output information that helps debug the source of the error. Potentially we can also insert a breakpoint instruction at the end of the fault handler.

Section 12.5 ("Analyzing Faults") of this book explains a number of ways to debug the issues. Very often it is very useful to generate a disassembled code listing of the project so that we can map the extract program address values to different part of the program easily. For example, when a fault exception occurred, the Program Counter (PC) value is pushed into the stack. We can extract the stacked PC using the method shown in Figure 12.4.

One we have extracted the stacked PC value, we can correlate this value to the program code using the disassembled code listing, as shown in Figure I.1.

With ARM® Compilation toolchain, you can generate disassembled code listing using fromelf:

```
$K\ARM\ARMCC\BIN\fromelf -c -d -e -s #L --output list.txt
```

If you are using Keil™ MDK 4.6x or 4.7x, you can specific a command line to be executed after the compilation is done. This can be achieved using "User" tab in the project option, and add the following command line in "Run User Programs After Build/Rebuild":

```
$K\ARM\BIN40\fromelf -c -d -e -s #L --output list.txt
```

Or if you are using Keil MDK 4.5x or older version, the command line is

```
$K\ARM\BIN40\fromelf -c -d -e -s #L --output list.txt
```

For users of the GNU Compiler, you can create disassembled listing using:

```
arm-none-eabi-objdump -S <filename>.o > list.txt
```

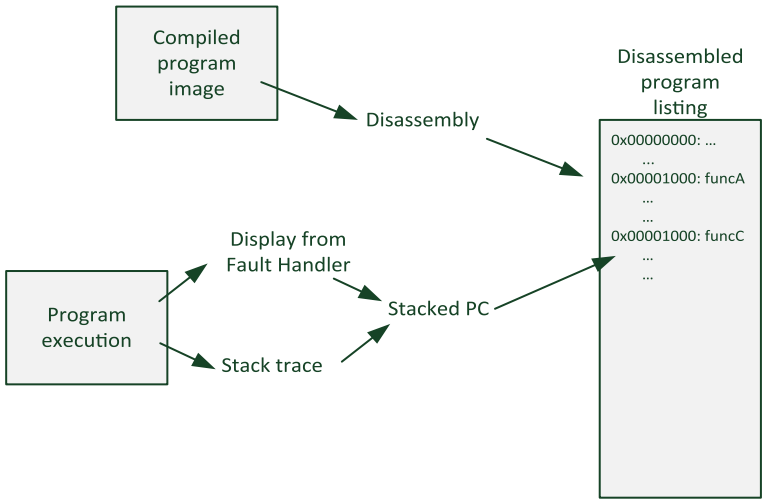

FIGURE I.1

Using disassembled code listing to locate faulting location

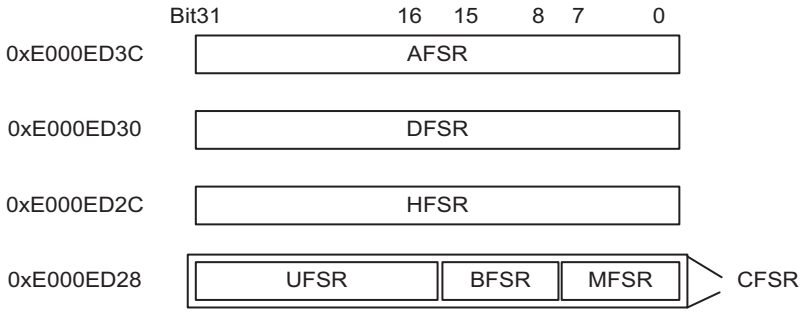

FIGURE I.2

Fault Status Registers

Once you have generated the disassembled program listing, it is much easier to understand the program during debugging, single stepping, and so on.

In addition to the stacked PC value, there are various fault status registers available (see section 12.4). The MMSR, BFSR, and UFSR registers can be accessed in one go using a word transfer instruction. In this situation, the combined fault status register is called the Configurable Fault Status Register (CFSR) (see Figure I.2).

For users of CMSIS-compliant device drivers, these fault status registers can be accessed using the following symbols:

```
SCB->CFSR: Configurable Fault Status register
SCB->HFSR: Hard Fault Status register
SCB->DFSR: Debug Fault Status register
SCB->AFSR: Auxiliary Fault Status register
```

In addition, you can also access fault address registers:

```
SCB->BFAR : Bus Fault Address register
SCB->MMFAR: MemManage Fault Address Register
```

Remember to use correct access sequence when using fault address registers:

1. Read BFAR/MMAR.
2. Read BFARVALID/MMARVALID. If it is 0, the BFAR/MMAR read should be discarded.
3. Clear BFARVALID/MMARVALID.

The reason for this procedure instead of reading valid bits first is to prevent a fault handler being pre-empted by another higher-priority fault handler after the valid bit is read, which could lead to the following erroneous fault-reporting sequence:

1. Read BFARVALID/MMARVALID.
2. Valid bit is set, going to read BFAR/MMAR.
3. Higher-priority exception pre-empts existing fault handler, which generates another fault, causing another fault handler to be executed.
4. The higher-priority fault handler clears the BFARVALID/MMARVALID bit, causing the BFAR/MMAR to be erased.
5. After returning to the original fault handler, the BFAR/MMAR is read, but now the content is invalid and leads to incorrect reporting of the fault address.

Therefore, it is important to read the BFARVALID/MMARVALID after reading the Fault Address register to ensure that the address register content is valid.

After the fault reporting is done, the fault status bit in the FSR should be cleared so that next time the fault handler is executed, the previous faults will not confuse the fault handler. In addition, if the fault address valid bit is not clear, the Fault Address register will not get an update for the next fault.

For users who are developing safety critical system, an assembly wrapper handler for fault handlers is useful. This is because a fault could be caused by stack pointer corruption, and usually in the beginning of C functions a number of registers would be pushed to the stack. So the PUSH operation in the beginning of the C handler could make things worst. As we know that that R0-R3 and R12 should have been pushed to the stack in the beginning of the exception handler, we can use these registers to carry out a stack point check operation to ensure that the stack pointer is still pointing the stack in the right memory address range, before we branch to the handler in C.

I.6 Understanding the cause of the fault

After obtaining the information we need, we can establish the cause of the problem. Tables I.1—I.5 listed some of the common reasons that faults occur.

Table I.1 MemManage Fault Status Register

Bit	Possible Causes
MMARVALID (bit 7)	Indicates the Memory Manage Address Register (MMAR) contains a valid fault addressing value.
MLSPERR (bit 5)	Error occurred during lazy stacking (deferred stacking of floating point registers) 1) Floating Point Context Address Register (FPCAR) is corrupted 2) MPU setting changed during execution of an exception handler. Note: If a SP corruption or stack overflow occurred, the fault exception would have been triggered with MSTKERR bit set, before lazy stacking taking place.
MSTKERR (bit 4)	Error occurred during stacking (starting of exception) 1) Stack pointer is corrupted. 2) Stack size go too large, reaching a region not defined by the MPU or disallowed in the MPU configuration.
MUNSTKERR (bit 3)	Error occurred during unstacking (ending of exception). If there was no error stacking but error occurred during unstacking, it might be: – Stack pointer was corrupted during exception. – MPU configuration changed by exception handler.
DACCVIOL (bit 1)	Violation to memory access protection, which is defined by MPU setup. For example, user application (unprivileged) trying to access privileged-only region.
IACCVIOL (bit 0)	1) Violation to memory access protection, which is defined by MPU setup. For example, user application (unprivileged) trying to access privileged-only region. Stacked PC might able to locate the code that has caused the problem. 2) Branch to non-executable regions, which could be caused by – simple coding error. – use of incorrect instruction for exception return – corruption of stack, which can affect stacked LR which is used for normal function returns, or corruption of stack frame which contains return address. – Invalid entry in exception vector table. For example, loading of an executable image for traditional ARM processor core into the memory, or exception happen before vector table in SRAM is set up.

Table I.2 Bus Fault Status Register

Bit	Possible Causes
BFARVALID (bit 7)	Indicate the Bus Fault Address Register contain a valid bus fault address.
LSPERR (bit 5)	Error occurred during lazy stacking (deferred stacking of floating point registers) – Floating Point Context Address Register (FPCAR) is corrupted Note: If a SP corruption or stack overflow occurred, the fault exception would have been triggered with STKERR bit set, before lazy stacking taking place.
STKERR (bit 4)	Error occurred during stacking (starting of exception) – Stack pointer is corrupted. – Stack size go too large, reaching an undefined memory region. – PSP is used but not initialized.
UNSTKERR(bit 3)	Error occurred during unstacking (ending of exception). If there was no error stacking but error occurred during unstacking, it might be that the stack pointer was corrupted during exception.
IMPRECISERR (bit 2)	Bus error during data access. Bus error could be caused by – Coding error that leads to access of invalid memory space. – Device accessed return error response. For example, device has not been initialized, access of privileged-only device in user mode, or the transfer size is incorrect for the specific device.
PRECISERR (bit 1)	Bus error during data access. The fault address may be indicated by BFAR. A bus error could be caused by: – Coding error which leads to access of invalid memory space. – Device accessed return error response. For example, the device has not been initialized, or the transfer size is incorrect for the specific device, or access of privileged-only device in unprivileged state (including System Control Space registers and debug components on System region).
IBUSERR (bit 0)	Branch to invalid memory location, which could be caused by – simple coding error. For example, caused by incorrect function pointers in program code. – use of incorrect instruction for exception return – corruption of stack, which can affect stacked LR which is used for normal function returns, or corruption of stack frame which contains return address. – Invalid entry in exception vector table. For example, loading of an executable image for traditional ARM processor core into the memory, or exception happen before vector table in SRAM is set up.

Table I.3 Usage Fault Status Register

Bit	Possible Causes
DIVBYZERO (bit 9)	Divide by zero taken place and DIV_0_TRP is set. The code causing the fault can be located using stacked PC.
UNALIGNED (bit 8)	Unaligned access attempted with multiple load/store instruction, exclusive access instruction or when UNALIGN_TRP is set. The code causing the fault can be located using stacked PC.
NOCP (bit 3)	Attempt to execute a floating point instruction when the Cortex-M4 floating point unit is not available or when the floating point unit has not been enabled. Attempt to execute a coprocessor instruction. The code causing the fault can be located using stacked PC.
INVPC (bit 2)	1) Invalid value in EXC_RETURN number during exception return. For example, – Return to thread with EXC_RETURN = 0xFFFFFFF1 – Return to handler with EXC_RETURN = 0xFFFFFFF9 To investigate the problem, the current LR value provides the value of LR at the failing exception return. 2) Invalid exception active status. For example: – Exception return with exception active bit for the current exception already cleared. Possibly caused by use of VECTCLRACTIVE, or clearing of exception active status in SCB->SHCSR. – Exception return to thread with one (or more) exception active bit still active. 3) Stack corruption causing the stacked IPSR to be incorrect. For INVPC fault, the Stacked PC shows the point where the faulting exception interrupted the main/pre-empted program. To investigate the cause of the problem, it is best to use exception trace feature in ITM. 4) ICI/IT bit invalid for current instruction. This can happen when a multiple-load/store instruction gets interrupted and, during the interrupt handler, the stacked PC is modified. When the interrupt return takes place, the non-zero ICI bit is applied to an instruction that do not use ICI bits. The same problem can also happen due to corruption of stacked PSR.
INVSTATE (bit 1)	1) Loading branch target address to PC with LSB equals zero. Stacked PC should show the branch target. 2) LSB of vector address in vector table is zero. Stacked PC should show the starting of exception handler. 3) Stacked PSR corrupted during exception handling, so after the exception the core tries to return to the interrupted code in ARM state.
UNDEFINSTR (bit 0)	1) Use of instructions not supported in Cortex-M3/M4 2) Bad/corrupted memory contents 3) Loading of ARM object code during link stage. Check compilation steps. 4) Instruction align problem. For example, if GNU Toolchain is used, omitting of .align after .ascii might cause next instruction to be unaligned (start in odd memory address instead of half-word addresses).

Table I.4 Hard Fault Status Register

Bit	Possible Causes
DEBUGEVT (bit 31)	Fault is caused by debug event: 1) Breakpoint/watchpoint events. 2) If the HardFault handler is executing, it might be caused by execution of BKPT without enable monitor handler (MON_EN=0) and halt debug not enabled (C_DEBUGEN=0). By default some C Compilers might include semi-hosting code that use BKPT.
FORCED (bit 30)	1) Trying to run SVC/BKPT within SVC/monitor or another handler with same or higher priority. 2) A fault occurred, but it corresponding handler is disabled or cannot be started because another exception with same or higher priority is running, or because exception mask is set.
VECTBL (bit 1)	Vector fetch failed. Could be caused by: 1) Bus fault at vector fetch. 2) Incorrect vector table offset setup.

Table I.5 Debug Fault Status Register

Bit	Possible Causes
EXTERNAL (bit 4)	EDBGRQ signal has been asserted.
VCATCH (bit 3)	Vector catch event has occurred.
DWTTRAP (bit 2)	DWT watchpoint event has occurred.
BKPT (bit 1)	1) Breakpoint instruction is executed 2) FPB unit generated a breakpoint event. In some cases BKPT instructions are inserted by C startup code as part of the semi-hosting debugging setup. This should be removed for a real application code. Please refer to your compiler document for details.
HALTED (bit 0)	Halt request in process or debug control

I.7 Other possible problems

A number of other common problems are in Table I.6.

Table I.6 Other Possible Problems

Situations	Possible Causes
No program execution	Vector table could be setup incorrectly. – Located in incorrect memory location. – LSB of vectors (including hard fault handler) is not set to 1. – Use of branch instruction (as in vector table in traditional ARM processor) in the vector table because incorrect startup code was used. Please generate a disassembly code listing to check if the vector table is setup correctly.
Program crashes after a few numbers of instructions.	Possibly caused by incorrect endian setting, or Incorrect Stack Pointer setup (check vector table), or Use of C object library for traditional ARM processor (ARM code instead of Thumb code). The offending C object library code could be part of the C startup routine. Please check compiler and linker options to ensure that Thumb or Thumb-2 library files are used.
Processor does not enter sleep when WFE is executed	A WFE instruction does not always result in sleep. If the internal event register was set before the WFE instruction, it will clear the event register and act as NOP. Therefore in normal coding WFE should be used with a loop.
Processor stop executing unexpectedly	When Sleep-on-exit feature is enabled, the processor enter sleep mode when return from exception handler to Thread mode, even if no WFI or WFE instructions are used.
Unexpected SEVONPEND behavior	The SEVONPEND in the System Control Register (SCB->SCR) enable a disabled interrupt to wake up the processor from WFE, but not WFI. The wake-up event is generated only at a new pending of an interrupt. If the interrupt pending status was already set before execution of WFE, arrival of a new interrupt request will not generate the wake-up event and hence will not wake up the processor.
Interrupt priority level not working as expected	Unlike many other processors, the Cortex-M processors use value 0 for highest programmable exception priority level. The larger the priority-level value, the lower the priority is. When programming the priority-level registers for interrupt, make sure the priority values are written to the implemented bits of the registers. The least significant bits of the priority-level registers are not implemented. In most Cortex-M3/M4 microcontrollers, it is either 3-bits (8 levels) or 4-bits (16 levels). When there is less than 8-bit of priority level, the LSBs are not implemented.

(*Continued*)

Table I.6 Other Possible Problems—Cont'd

Situations	Possible Causes
	So if you write priority-level values like 0x03, 0x07, etc., to these registers, the value will become 0x00.
SVC instruction result in fault exception	The Cortex-M processors do not support recursive exception – an exception cannot pre-empt unless it is higher priority than the current level. As a result, you cannot use SVC within a SVC, HardFault or NMI handler, or any other exception handler that has the same or higher priority than the SVC exception.
Parameters passing to SVC corrupted	When passing parameter to exception handlers like SVC, the extraction of parameters (R0–R3) should be carried by getting the parameters from the stack frame instead of using values on the register bank. This is because there could be a chance that another exception was processed just before entering the SVC (new arrival exception case). Since the other exception handler can change R0-R3 (AAPCS does not require a C function to keep R0-R3, R12 unchanged), the values of R0-R3 and R12 can be undefined when entering the SVC handler. To obtain the parameters correctly, the stacked data should be used. This involves using a simple SVC wrapper in assembly to extract the correct stack pointer and pass it on to the C handler as a C pointer. Example code of this can be found in section 10.3. Similar arrangement can be found in returning data from exception handler to the interrupted program. The handler should store the return data into the stack frame. Otherwise, the value in the register bank will be overwritten during unstacking.
SysTick exception occur after clearing TICKINT	The TICKINT bit in the SysTick Control and Status Register enable and disable the generation of SysTick exception. However, if the SysTick exception is already in pending state, clearing of TICKINT will not stop the SysTick exception from getting fired. To ensure the SysTick exception will not be generated, you need to clear TICKINT and the SysTick pending status in the Interrupt Control and Status Register (SCB->ICSR).
JTAG locked out	In many Cortex-M based microcontrollers the JTAG and I/O pins are shared. If the I/O functions for these pins are enabled right in the start of the program, you could find that your will not able to debug or erase the flash again.
Unexpected extra interrupt	Some microcontrollers as a write buffer in the bus bridge for the peripheral bus. This makes it possible for an exception handler to clear a peripheral interrupt by writing to the peripherals, exit the handler, and then enter the interrupt again as the peripheral cannot de-assert the

Table I.6 Other Possible Problems—Cont'd

Situations	Possible Causes
	interrupt request fast enough. There are several workarounds for this problem: – The interrupt service routine (ISR) could carry out a dummy access to the peripheral before exception exit. However, this can increase the duration of the ISR. – The clearing of the interrupt can be moved the beginning of the ISR to that the interrupt is cleared before ISR end. This might not work if the ISR duration is shorter than the buffered write delay, so extensive testing should be carried out for various peripheral clock ratios.
Problem with using normal interrupt as software interrupt	Some users might intend to use unassigned exception types available in NVIC for software interrupt functions. However, the external interrupt and the PendSV exceptions are imprecise. That means the exception handler might not happen immediately after pending it. To handle this, a polling loop can be used to ensure the exception is carried out.

Unexpected BLX or BX instructions which switch to ARM state	For users of GNU assembler, if a function symbol is from a different file, you need to ensure the function name is declared as a function:
	`.text`
	`.global my_function_name`
	`.type my_function_name, %function`
	Otherwise, a branch or call to this function might results in accidental switching to ARM state.
	In addition, during linking stage, you might need "`-mcortex-m3 -mthumb`" options for the GNU linker. Otherwise, the GNU linker might pull in the wrong version of C library code.

Unexpected disabling of interrupt	The behavior of the Cortex-M and ARM7TDMI processors are different regarding exception returns. In ARM7TDMI, if the interrupt is disabled inside an interrupt handler, it will be automatically re-enabled at exception return due to restore of CPSR (I bit). For the Cortex-M processor, if you disable interrupts manually using PRIMASK (e.g., "CPSID I" instruction or __disable_irq()), you will need to re-enabling it at later stage inside the interrupt handler. Otherwise, the PRIMASK register will remain set after exception return and will block all interrupt from being taken place.
Unexpected unaligned accesses	The Cortex-M3/M4 processor supports unaligned transfers on single load/store. Normally C compilers do not generate unaligned transfer except for packed structures and manual manipulation of pointers.

(*Continued*)

Table I.6 Other Possible Problems—Cont'd

Situations	Possible Causes
	For programming in assembly language, it could be a bigger issue as users might accidentally use unaligned transfers and not know it. For example, when a program reading a peripheral with word transfer of an unaligned address, the lowest two bit of the addresses might be ignored when using other ARM processors (as the AHB to APB bridge might force these two bits to 0). In the case of the Cortex-M3/M4 processor, if the same software code is used, it will divide the unaligned transfer into multiple aligned transfers. This could end up with different results. Equally, issues can be found when porting software for the Cortex-M3/M4 processor to other ARM processors that do not support unaligned transfers. It is easy to detect if the software generate unaligned transfers. This can be done by setting the UNALIGN_TRP bit in the Configuration Control Register (SCB->CCR) in the System Control Block. By doing this a usage fault will be triggered when an unaligned transfer happened and you can then eliminate all unexpected unaligned transfers.