Diagnosing Software Faults in Stellaris® Microcontrollers

Joe Kroesche, Stellaris® Software

ABSTRACT
During typical development efforts, system operation can sometimes end up in a fault handler. At first glance, faults might seem cryptic or difficult to understand, but Stellaris microcontrollers include several features to help determine why a fault occurred. Read this application note to learn about the fault handling capabilities of Stellaris microcontrollers, how to diagnose faults, and how to prevent them.

Contents
1 Introduction ................................................................................................................................. 1
2 Fault System Overview .................................................................................................................. 1
3 Diagnosing Faults ......................................................................................................................... 4
4 Typical Faults and How to Avoid Them ............................................................................................ 15
5 General Fault Debug Flow ............................................................................................................ 18
6 Conclusion ................................................................................................................................ 18
7 References ................................................................................................................................ 18

1 Introduction
When software running on a Stellaris Cortex™-M processor encounters an error, it generates a fault, and the program execution is directed to a fault handler. In some cases, fault occurrence in a program can seem mysterious and difficult to debug. However, a fault should not be the cause of debug trepidation. The Stellaris Cortex-M processor provides a powerful fault handling system and several features to help you find the cause of a fault. After debugging a few faults, you will be more comfortable handling and troubleshooting faults as they happen, and will be able to find the cause in a few minutes.

2 Fault System Overview
In a Stellaris Cortex-M processor, a fault is a type of exception. There are 15 system exceptions including SysTick and SVC. Four of these system exceptions are faults, with one for each type of fault that can occur in the processor when it is running code.

Table 1 shows the list of fault types and their causes.

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Description</th>
<th>Priority Level</th>
<th>Required?</th>
<th>If not enabled...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard fault</td>
<td>A hard fault includes any fault that is not covered by the following three categories. Any unhandled fault escalates to a hard fault.</td>
<td>Highest priority, fixed at -1&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Required</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Memory Management fault</td>
<td>A memory management fault occurs when executing code attempts to access an illegal location or violates a rule of the Memory Protection Unit (MPU).</td>
<td>Configurable, lower than Hard fault</td>
<td>Optional</td>
<td>Escalates to a Hard fault</td>
</tr>
</tbody>
</table>

<sup>a</sup> Highest priority, fixed at -1.
### Table 1. Fault Types (continued)

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Description</th>
<th>Priority Level</th>
<th>Required?</th>
<th>If not enabled...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus fault</td>
<td>A bus fault occurs when there is an error on the bus that happens when accessing a peripheral or memory.</td>
<td>Configurable, lower than Hard fault</td>
<td>Optional</td>
<td>Escalates to a Hard fault</td>
</tr>
<tr>
<td>Usage fault</td>
<td>A usage fault occurs when there is a program error such as an illegal instruction, alignment problem, or attempt to access a non-existent co-processor.</td>
<td>Configurable, lower than Hard fault</td>
<td>Optional</td>
<td>Escalates to a Hard fault</td>
</tr>
</tbody>
</table>

- a. Only the reset and non-maskable interrupt (NMI) exceptions are higher in priority than the hard fault.

The Hard fault has the highest priority of any fault in the system. The priority of the fault is fixed at -1, and the only higher priority exceptions in the system are reset and non-maskable interrupt (NMI). The remaining fault types have configurable priority levels but are typically lower priority than the Hard fault and higher priority than every other type of exception or interrupt. Only the Hard fault is required and cannot be disabled. The other three fault types can be enabled or disabled in the system. If one of the other three fault types occurs and is not enabled, then it is escalated to a Hard fault.

See the documentation from the “References” section for more details about fault types and how they can be configured.

When a fault occurs, there are two useful tools available to find the cause: the exception stack frame (see “Exception Stack Frame” for more information) and the fault status registers (see “Fault Status Registers” for more information).

#### 2.1 Exception Stack Frame

The Cortex-M processor uses a uniform mechanism for saving program context information on the stack. This same mechanism is used for all types of exceptions that can occur in the system, including faults and interrupts. When an exception occurs (including a fault), the following registers are pushed on the stack, shown here in order of increasing address on the stack:

```
SP+00 R0
SP+04 R1
SP+08 R2
SP+12 R3
SP+16 R12
SP+20 LR
SP+24 PC
SP+28 xPSR
```

The register values saved in the exception stack frame provide a snapshot of the processor context at the time the exception occurred. For a fault handler that is written in C, the compiler saves onto the stack any additional registers that are used by the fault handling function, so that the entire system context is saved in a fault handler.

The saved value of the working registers (R0-R3, R12) can be useful, especially if you isolate the problem to an instruction that uses one of those registers. These values allow you to reconstruct the condition of the instruction at the time the instruction executed. The Program Counter (PC) and Link Register (LR) values are the most useful information that is saved in the exception stack frame.

First, the saved value of the PC often points to the offending instruction. Using the debugger to examine the instruction at that address along with the register values often provides enough clues that the cause of the fault becomes apparent. Sometimes it is not that easy though, and the saved PC only points in the vicinity of the problem, or does not provide any useful information at all (this is rare).
Second, the saved value of the LR can often be used to provide you with some context about what your program was doing when the fault occurred. For example, if the code was executing in a certain function when the fault occurred, the saved LR might point back into the function that called it. By examining the code at the location pointed to by the saved LR, you might be able to tell more about the calling function.

The saved LR can also be used, in conjunction with the rest of the stack, to unwind a series of function calls, providing an even better history. However, this process is a lot of trouble to do manually without an automatic stack backtrace from the debugger and is usually only necessary in tricky debug situations.

Sometimes the exception stack frame is not available. This usually happens when the stack pointer is pointing to a non-RAM location, which might be due to stack corruption or overflow.

2.2 Fault Status Registers

There are three fault status registers, one for each type of fault:

- Memory Management Fault Status (MFAULTSTAT) register (8-bits at 0xE000.ED28)
- Bus Fault Status (BFAULTSTAT) register (8-bits at 0xE000.ED29)
- Usage Fault Status (UFAULTSTAT) register (16-bits at 0xE000.ED2A)

Each of these fault status registers contains status bits to indicate the cause of the fault. Also, for the memory management and bus fault types, there is an address register that holds the address that caused the fault. The three status registers can be read at separate addresses as shown above, but typically are read as a combined 32-bit word from address 0xE000.ED28.

2.2.1 Memory Management Fault Status (MFAULTSTAT) Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>MMARVALID</td>
<td>Valid fault address is stored in the Memory Management Address Register (MMADR).</td>
</tr>
<tr>
<td>6:5</td>
<td>unused</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>MSTKERR</td>
<td>Memory access violation occurred while stacking on exception entry.</td>
</tr>
<tr>
<td>3</td>
<td>MUNSTKERR</td>
<td>Memory access violation occurred while unstacking on exception return.</td>
</tr>
<tr>
<td>2</td>
<td>unused</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>DACCVIOL</td>
<td>Data access violation.</td>
</tr>
<tr>
<td>0</td>
<td>IACCVIOL</td>
<td>Instruction access violation.</td>
</tr>
</tbody>
</table>

Bit 7 is set if the Memory Management Fault Address Register (MMADR), located at 0xE000.ED34, contains the address that caused the memory management fault.

2.2.2 Bus Fault Status (BFAULTSTAT) Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>BFARVALID</td>
<td>Valid fault address is stored in Bus Fault Address Register (FAULTADDR).</td>
</tr>
<tr>
<td>6:5</td>
<td>unused</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>STKERR</td>
<td>Bus fault occurred while stacking on exception entry.</td>
</tr>
<tr>
<td>3</td>
<td>UNSTKERR</td>
<td>Bus fault occurred while unstacking on exception return.</td>
</tr>
<tr>
<td>2</td>
<td>IMPRECISERR</td>
<td>Stacked PC does not indicate exact address of bus fault.</td>
</tr>
<tr>
<td>1</td>
<td>PRECISERR</td>
<td>Stacked PC indicates exact address of bus fault.</td>
</tr>
<tr>
<td>0</td>
<td>IBUSERR</td>
<td>Instruction bus error.</td>
</tr>
</tbody>
</table>

Bit 7 is set if the Bus Fault Address Register (FAULTADDR), located at 0xE000.ED38, contains the address that caused the bus fault.
2.2.3 Usage Fault Status (UFAULTSTAT) Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:10</td>
<td>unused</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>DIVBYZERO</td>
<td>Traps an instruction that attempted to divide-by-zero. This trap can be enabled or disabled.</td>
</tr>
<tr>
<td>8</td>
<td>UNALIGNED</td>
<td>Traps an attempt to make an unaligned memory access. This trap can be enabled or disabled.</td>
</tr>
<tr>
<td>7:4</td>
<td>unused</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>NOCP</td>
<td>Attempted to execute a coprocessor instruction.</td>
</tr>
<tr>
<td>2</td>
<td>INVPC</td>
<td>Attempted exception return caused illegal load of PC.</td>
</tr>
<tr>
<td>1</td>
<td>INVSTATE</td>
<td>Invalid processing/instruction state.</td>
</tr>
<tr>
<td>0</td>
<td>UNDEFINSTR</td>
<td>Attempt to execute an undefined instruction.</td>
</tr>
</tbody>
</table>

3 Diagnosing Faults

Here is a common debugging scenario:
1. Developer discovers the application stops running.
2. Using a debugger, developer discovers that the application ends up in a fault handler.
3. Developer gets stuck not knowing what to do and asks for support.

The goal of this application note is to demystify step 3 and provide the developer with techniques to use to get the application running again.

The remainder of this section presents different fault scenarios and shows how to use the Stellaris features (described in the previous section) to determine the cause of the fault.

3.1 Default Interrupt Handler

Technically, this is not a fault. However, this common scenario appears in customer support requests to the Stellaris applications team. The following screen capture shows a program that is halted in the IntDefaultHandler() function.

```c
static void IntDefaultHandler(void)
{
    // Go into an infinite loop.
    while (1)
    {
        
    }
}
```

Once here, it is safe to assume that an interrupt occurred that does not have a handler function entered in the vector table. Once it is known that the program has entered the default handler, it is often obvious which interrupt has triggered and has no handler. In this case, there is no further debugging required. However, sometimes you want to find out which peripheral caused this interrupt to happen. The quickest way to find
out is to look at the value of the **Program Status Register (xPSR)** register. The lower 8 bits contain the number of the current exception.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Registers</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>0x00000932</td>
</tr>
<tr>
<td>SP</td>
<td>0x200000C8</td>
</tr>
<tr>
<td>LR</td>
<td>0xFFFFFFF9</td>
</tr>
<tr>
<td>xPSR</td>
<td>0x01000023</td>
</tr>
<tr>
<td>R0</td>
<td>0x40031000</td>
</tr>
<tr>
<td>R1</td>
<td>0x00000001</td>
</tr>
<tr>
<td>R2</td>
<td>0x00000000</td>
</tr>
<tr>
<td>R3</td>
<td>0x00000101</td>
</tr>
</tbody>
</table>

In the above example, the lower 8 bits of the xPSR register show the exception number is 0x23 (or 35 decimal).

Referring to the Interrupts table in the “Exception Model” section of the “Cortex-M Processor” chapter of a Stellaris data sheet, shows that vector number 35 is Timer 0A.

<table>
<thead>
<tr>
<th>Vector</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>0x0000.0094</td>
</tr>
<tr>
<td>34</td>
<td>0x0000.0088</td>
</tr>
<tr>
<td>35</td>
<td>0x0000.008C</td>
</tr>
<tr>
<td>38</td>
<td>0x0000.0090</td>
</tr>
<tr>
<td>37</td>
<td>0x0000.0094</td>
</tr>
<tr>
<td>38</td>
<td>0x0000.0096</td>
</tr>
</tbody>
</table>

Now look in the vector table (usually in the startup code) at the vector for Timer0A.

The above screen capture shows that the vector entry for Timer 0A is the `IntDefaultHandler()` function which is the default when no handler has been provided. In this case, the correct handler function was not entered in the table. The screen capture below shows the correct handler function entered in the vector table.
Another way to determine which interrupt caused the processor to enter the default handler is to examine the NVIC Active Interrupt registers to see what interrupt is active, as shown below:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIC_PEND0</td>
<td>0x00200000</td>
</tr>
<tr>
<td>NVIC_PEND1</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NVIC_UNPEND0</td>
<td>0x02000000</td>
</tr>
<tr>
<td>NVIC_UNPEND1</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NVIC_ACTIVE0</td>
<td>0x00080000</td>
</tr>
<tr>
<td>NVIC_ACTIVE1</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NVIC_PRI0</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NVIC_PRI1</td>
<td>0x00000000</td>
</tr>
<tr>
<td>NVIC_PRI2</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

In the above screen capture, the debugger provides a view to see the value of this register. If your debugger does not provide such a view, you can examine these registers using a memory viewer starting at address 0xE000.E300. In this case, bit 19 of the first Active Interrupt register is set, indicating that interrupt number 19 is active and is likely the cause of the interrupt. If more than one bit is set in this register, then the highest priority interrupt is the active interrupt.

Refer again to the Interrupt table in the "Exception Model" section of a Stellaris data sheet and look up interrupt number 19. As before, this shows that the active interrupt is Timer0A.

3.1.1 Summary
1. If your program ends up in the `IntDefaultHandler()`, it is probably because the handler function was not entered in the vector table (see startup code source file).
2. Use the debugger to examine the lower 8 bits of the xPSR register while the program is halted in the `IntDefaultHandler()` function, and obtain the interrupt vector number.
3. Look up the vector number in the Stellaris data sheet to find out which peripheral caused the interrupt.
4. (optional) Examine the NVIC Active Interrupt registers to determine the highest priority active interrupt and look that up in the Stellaris data sheet.

3.2 Bus Fault Scenario 1

In this common scenario, when using a debugger, the programmer finds that the program has entered the Hard Fault handler.

```c
static void FaultISR(void)
{
    // Enter an infinite loop.
    //
    while(1)
    {
    }
    }
```
In this example, use the debugger to examine the fault status register. Your debugger may provide a viewer for this register (as shown below), or you can use a memory viewer to examine address 0xE000.ED28.

In this case, the NVIC Fault Status (NVIC_FAULT_STAT) register has a value of 0x0000.8200. Refer to any one of the documents listed in the "References" section, or to "Diagnosing Faults", to see the meaning of the bits in this register. The value shown here indicates a bus fault, and the bits that are set are BFARVALID and PRECISERR. This means that the Bus Fault Address Register (FAULTADDR) register contains the exact value of the address that triggered the bus fault.

Use the debugger to view the value of the FAULTADDR register. If the debugger does not have a viewer for this register, you can examine the memory at address 0xE000.ED38.

The value in the NVIC Fault Address (NVIC_FAULT_ADDR) register is 0x4003.100C. Refer to the Memory Map table in the "Memory Model" section of the "Cortex-M Processor" chapter of the Stellaris data sheet, which shows that the faulting address is in the register space of the Timer 1 peripheral.
This means that some part of the program was trying to access Timer 1 and caused a bus fault. 

**Note:** This is a common support scenario for the Stellaris applications team. When there is a bus fault and the faulting address is in the register space of a peripheral, it is almost always because the programmer did not enable the peripheral before trying to use it.

Consider the following code snippet. The programmer enabled Timer 0, but forgot to add an additional line of code to enable Timer 1.

```c
// Enable the peripherals used by this example.
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);

// Enable processor interrupts.
IntMasterEnable();
```

The following shows the correct code. This eliminates the fault.

```c
// Enable the peripherals used by this example.
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER1);

// Enable processor interrupts.
IntMasterEnable();
```

At this point, the cause of this fault has been found and corrected and no further debugging is required. However, another path could be taken to find the problem, and that is to start with the exception stack frame. Use the debugger to find the value of the stack pointer.
If your fault handler function has any code besides a simple infinity loop, then it may use some stack space. You must adjust the value of the stack pointer by this amount in order to find the start of the exception stack frame. For StellarisWare® examples, the fault handler function is simple and the stack pointer points directly to the exception stack frame.

Use the debugger to view memory at this location.

In the screen capture above, the first two rows show the exception stack frame. The first row shows the saved values of registers R0-R3. The second row begins with the saved value of R12, followed by LR, PC, and then xPSR register. To start with, the most interesting value is the PC, which is 0x0000.0CBC. There is a high probability that this points either directly at, or near the instruction that caused the fault.

Use the debugger to disassemble at this address. It helps to start disassembling a few instructions earlier to provide some context for the disassembled code.

The instruction at 0x0000.0CBC is most likely the one that caused the fault. It is an LDR instruction, and is reading from the address in R0, plus an offset of 0xC. Refer back to the previous screen capture to see that the saved value of R0 was 0x4003.1000, and adding 0xC gives 0x4003.100C. The processor was trying to read from 0x4003.100C and this is likely what caused the fault. You can see that this is the same conclusion...
that was found earlier in this section by examining the fault status register and the \texttt{FAULTADDR} register. So we have used a second method to find the same cause for the fault.

Because the cause of this fault has been found by two different methods, no further debugging is necessary. Even though this problem has been debugged, for the purpose of demonstration we can also used the saved value of \texttt{LR} to find a little more information. From a previous screen capture, the saved \texttt{LR} value was 0x0000.05A9. Again, use the debugger to disassemble code at this address.

There are several things to note about this disassembled code. First, the \texttt{LR} was 0x5A9. The lower bit of the address is set and the address is odd. This is the way the ARM-based processor indicates it is running in Thumb mode. So the real address of the instruction is 0x5A8. The \texttt{LR} points to the address that executes on return, which means that the instruction that made the function call is the instruction before the \texttt{LR} address. In the screen capture above, this instruction is a call to the \texttt{TimerConfigure()} function, from the \texttt{main()} function indicating the location in the \texttt{main()} function that made a call to the \texttt{TimerConfigure()} function, causing the bus fault.

So even though it was not necessary for this example to debug this far, these methods demonstrate a way to obtain more information about what your program is doing when a fault occurs and helps you to trace back to the source of the problem. The saved \texttt{LR} may not always provide a useful backtrace, but it can often be used this way to go back a level in the program.

\subsection{3.2.1 Summary}

If your program ends up in the hard fault handler, \texttt{FaultISR()}, there are two methods to use to find the cause (and sometimes both are needed).

\textbf{Method 1}

1. Use the debugger to examine the \texttt{NVIC_FAULT_STAT} register to find the type of fault and the status bits that indicate the specific cause.
2. If there is a valid fault address register (\texttt{FAULTADDR} or \texttt{MMADR}), then read that to find the faulting address.
3. Study the memory map in the Stellaris data sheet to find a clue about the cause of the fault. Often the address is in the register space of a peripheral.
4. Use this information to go back to the source code and try to identify the section of code that is causing the problem.

\textbf{Method 2}

1. Use the debugger to find the value of the stack pointer.
2. Adjust the value of the stack pointer if your fault handler uses the stack, otherwise, use the value as it is.
3. Use the debugger to examine eight words of the exception stack frame that are pointed to by the value of the stack pointer (SP).
4. Find the saved value of the PC and disassemble code at that address to try to find an instruction that may have caused the fault. Use the values of the saved registers to reconstruct the instruction parameters.

5. If further context is required, disassemble code pointed at by the saved value of the . Often this reveals the previous function in the call stack and helps to identify the cause of the fault.

3.3 Bus Fault Scenario 2

In this scenario, like “Bus Fault Scenario 1”, the programmer finds that the program is entering the hard fault handler, FaultISR().

As before, use the debugger to examine the NVIC_FAULT_STAT register.

This time, the value in the NVIC_FAULT_STAT register is 0x0000.0400. The IMPRECISERR bit is set, which means that a bus fault occurred, but that the exact faulting address is not known.

In order to try to debug this, we will use the exception stack frame. Find the value of the stack pointer. This value may need to be adjusted if your fault handler uses the stack. For StellarisWare examples, the stack pointer in the fault handler does point to the exception stack frame. In this example, the stack pointer is 0x2000.00C8, so use the debugger to view memory at this location.

The first two rows in the screen capture above show the exception stack frame. The first row shows the saved values of registers R0-R3. The second row shows the saved value of R12, LR, PC, and xPSR.

Use the debugger to disassemble code at the location of the saved PC, 0x4D4.
The instruction that is pointed to by the saved PC is a function call to `IntMasterEnable()`. However, because this is an imprecise bus fault, we know that this is not actually the instruction that caused the fault, it must be something that happened earlier in the execution. Observe that the previous instruction is another function call to the `TimerLoadSet()` function. This call is most likely where the fault occurred.

Use the debugger to disassemble the `TimerLoadSet()` function and take a look near the end of this function. While the imprecise bus fault does not show the exact location of the fault, the value of the saved PC should be within a few instructions. Therefore, the fault must have occurred near the end of the `TimerLoadSet()` function.

The screen capture above shows the entire disassembly of the `TimerLoadSet()` function. Notice that the last instruction before the return is a storeimprecise (STR) instruction. Whenever there is an imprecise bus fault, it is a good idea to look for the most recent store instruction that was executed prior to the occurrence of the fault. In this example, this store instruction was writing to the location pointed to by R0, with an offset of 0x2C. By looking at the previous two disassembly screens, we can see that no code has modified R0 since this store instruction was executed and R0 still holds the same value it had at that time. Get the value of R0 either by viewing the current processor registers or looking at the stored value in the exception stack frame (screen capture above), which in this case is 0x4003.1000. As in the “Bus Fault Scenario 1”, consult the Stellaris data sheet to see that this address is in the register space of the Timer 1 peripheral.

Now that we suspect that the fault was caused by a write to a Timer 1 register, go back to look at the source code to try to find a call to the `TimerLoadSet()` function.
Here is a call to the TimerLoadSet() function. Looking back in the program, we see that there was no function call or other code to enable the Timer 1 peripheral, which is why the bus fault occurred.

This bus fault scenario is similar to the previous one, except that in this case there was an imprecise bus fault and the exact faulting address was not known. When this happens, it requires a little more work to find the cause of the fault than a precise error.

If the bus fault is due to a read instruction, it produces a precise error bus fault because the processor must wait for the read cycle to complete before it can continue. The invalid access occurs as the instruction executes and the exact location is known. When the fault is due to a write instruction, it is usually an imprecise error. This is because the processor initiates the write cycle, but can then continue to execute additional instructions before the actual invalid bus access (write) occurs. Therefore, the exact location is not known.

### 3.3.1 Summary

1. If a bus fault is an imprecise error, find the value of the saved PC from the exception stack frame.
2. Disassemble the program at the location of the saved PC. It is a good idea to start disassembling a few instructions prior to the saved PC.
3. Look back in the execution stream a few cycles, and look for an STR instruction. The most recent STR instruction that executed prior to the imprecise bus fault is most likely the cause of the fault.

### 3.4 Usage Fault Scenario (plus Memory Management Fault)

As with the other scenarios, this scenario starts with the programmer finding that the program has entered the fault handler. Use the debugger to examine the NVIC_FAULT_STAT register.

The NVIC_FAULT_STAT register has a value of 0x0002.1000, which indicates both a usage fault and a bus fault. The usage fault status bit INVSTATE is set, which means that the processor encountered a state error when it tried to execute an instruction.

The next step is to look at the stack pointer.
In this case, the value of the stack pointer is 0x1FFF.FFFF4, which is not a valid memory location. The reason this memory location is not valid is because the stack was too small and overflowed. Because the stack overflowed, it is likely that an invalid address was used on return from a function call, and the processor PC was set incorrectly causing the invalid state.

The other fault status bit that is set is the STKERR bus fault, which means that the processor encountered an error when it tried to store the exception stack frame when the first fault occurred.

In this example, the stack was located at the beginning of memory and when it overflowed, it went outside the bounds of SRAM, making it easy to spot the problem. However, depending on the tool chain used and other factors, the stack may not be located at the beginning of memory. If it is not at the beginning of memory and it overflows then it may overwrite some data, and code that changes that data could also corrupt the stack. This problem is more subtle but can be the source of usage faults because the processor may return from a function to an invalid address for the PC.

### 3.4.1 Summary
1. If a usage fault occurs, examine the NVIC_FAULT_STAT register to find the type of usage fault.
2. Examine the stack pointer to find the exception stack frame.
3. If the stack overflows or is corrupted, the exception stack frame may not be available.
4. For INVPC (invalid PC), INVSTATE (invalid state), or UNDEFINSTR (undefined instruction) usage, the most probable cause is stack overflow or stack corruption.

### 3.5 Memory Management Fault Scenario
This scenario makes use of the StellarisWare mpu_fault example, which uses the MPU. The MPU can be used to set up protected memory regions, and when the processor executes code that violates a protected memory region, a memory management fault occurs. Here is a screen capture showing the NVIC_FAULT_STAT register after the program enters the fault handler.

The value in the NVIC_FAULT_STAT register is 0x0000.0082, which means that the MMARVALID and DACCVIOL memory management status bits are set. These bits indicate that a data access violation occurred and the faulting address can be found in the Memory Management Fault Address Register (MMADR). The screen capture above also shows a value in the MMADR of 0x0000.0100, which is in Flash memory.
The next step is to examine the exception stack frame. The stack in this case is pointing at 0x2000.00A8. Here is a screen capture showing the exception stack frame.

```
Disassembly (MPUFaultHandler) Memory (1) X

0x200000a8 RAM

Hex 32 Bit - TI Style
```

```
0x200000a8 12345678 00000100 00000007 20000114
0x200000e8 00000198 000000b0 00000019 61000000
0x200000e0 00000000 00000000 FFFFFFD A4420001
0x200000d8 00000000 00000000 00000000 00000000
0x200000e0 86411200 90414108 22750c28 00000e27
0x200000f8 FFFFFFD 00000007
```

As before, the most interesting value at this point is the saved PC, which is 0x190. Disassemble the code at 0x190.

```
Disassembly (main + 0x90) Memory (1)

0x00000100: 6c28 STR R6, [R4]
0x00000102: 467e LDR R0, #c#CONL3
0x00000104: f44f7130 MOV.W R1, #255
0x00000108: 5608 STR R0, [R1]
0x0000010a: 6e48 MOV R6, R5
0x0000010c: 6621 LDR R1, [R4]
0x0000010e: a05b ADD R0, PC, #0x164
0x00000110: 0000 STR R0, [SP]
0x00000112: 2001 CMP R1, #0x1
```

This disassembly screen capture shows that the processor was trying to execute a store (STR) instruction. It was writing the value of R1 to the location of R0. The exception stack frame shows the value of R0 was 0x0000.0100, which is the same as the value in the MMADR. The processor was trying to write a value to Flash memory which has been protected by the MPU.

### 3.5.1 Summary
1. For a memory management fault, examine the fault status register.
2. If the DACCVIOL (data access violation) or IACCVIOL (instruction access violation) bit is set, the fault is due to the program violation of an MPU region access restriction.
3. If the MMARVALID bit is set, then read the MMADR to determine the faulting address.
4. Examine the stack pointer to determine the location of the exception stack frame.
5. Examine the exception stack frame to find the value of the saved PC.
6. Disassemble the program at the location of the saved PC to find the instruction that caused the fault.

### 4 Typical Faults and How to Avoid Them
This section covers the most common causes of faults as determined by support requests to the Stellaris applications team.
4.1 **Bus Fault Due to Disabled Peripheral**

This type of fault is most commonly seen in customer help requests regarding faults. Any attempt to access a disabled peripheral, including DriverLib APIs, results in a bus fault.

To avoid this fault, be sure to call the `SysCtlPeripheralEnable()` function for all peripherals used by your application.

4.2 **Various Faults Caused by Stack Overflow**

Stack problems are another common reason for support requests. Stack problems can be manifested in obscure ways including multiple faults and problems that do not appear at the time of the original stack error, but instead appear much later during program execution. These problems can be tricky to debug.

If your application is encountering a fault or multiple faults and the cause is not obvious, take a look at the stack allocation. Often it is helpful just to examine the value of the stack pointer when the fault occurs and compare it to the valid range for the stack pointer (from the map file). If the stack pointer is near the base address of the stack, then that is evidence that your stack might be too small. Try increasing the stack size by 50% or 100% to see if the problem disappears.

A function that is using a stack that has overflowed its bounds overwrites memory outside the space allocated for the stack. Often, this is storage for program variables. If this happens, the values of those variables become corrupted and the program will not run correctly. This may not create a fault until later.

Another issue that can occur when a function is running with an overflowed stack is that the function itself may use a variable that is in the space that has been overwritten by the stack. The function may write a value to that variable, destroying the value that was saved on the stack, and then either saved registers, or even the PC, might have the wrong value when the function returns. This error can cause strange behavior including returning to a wrong address.

4.3 **Stack Corruption Due to Buffer Overflow**

Sometimes a function uses an array that is allocated on the stack, and then in the function body, the array is written beyond its bounds. This error can have the effect of overwriting other automatic variables used in the function as well as overwriting saved registers and the return address. One example is a function that looks like this:

```c
void myfunc(void)
{
    char mystring[8];
    ...
    usprintf(mystring, "my formatted output %d\n", somevar);
    ...
}
```

In the above example, eight bytes were allocated on the stack for the character array. Later the `usprintf` function was used to store text characters in the array without considering the array bounds. The format string and the numerical value are written beyond the end of the space allocated for `mystring`, which overwrites any other stack variables, saved registers and return address, and possibly the stack context of the function that called this function.

A similar thing can happen if a numerical array is allocated on the stack and is then written from a for-loop. Perhaps the index limits of the for-loop match the array size when the programmer first created the function, but then later the for-loop index limit is increased without also increasing the size of the allocated array.

Here is another problem that happens more often than you think.
void myfunc(void)
{
    int myarray[1024];
    ...
}

In this case, the programmer allocated a huge array on the stack. Unless the stack was adjusted for an unusually large program, this will likely cause a stack overflow the first time this function is entered.

Note: Be careful with arrays allocated on the stack and with array bounds in general.

4.4 Bad PC Address

There are a couple of common pitfalls that can cause a bad PC address. Both usually occur when using a function pointer to call another function.

In the first example, the function pointer is called before it has been initialized:

void myfunc(void)
{
    void (*mypfn)(void);
    ...
    mypfn();
    ...
}

The compiler generates code to call the function through the function pointer, and if this pointer does not point to a function, the PC can end up anywhere. This error can cause any of the following:

• The processor jumps to valid code somewhere in the program and continues executing. The bug does not show up until later.
• The processor jumps to some location in memory that does not contain a valid instruction and a usage fault (probably) occurs.
• The processor jumps to a location outside the valid memory window and a bus fault and/or usage faults occur.

Another problem can happen when using a hard-coded address for the function pointer:

    mypfn = 0x100; // hard-coded address of jump destination
    ...
    mypfn();

This error causes an INVSTATE usage fault, which occurs because even though instructions are 16-bit aligned (always even address), the ARM-based processor core uses the lower bit of the destination address to determine the instruction mode: ARM or Thumb. The Cortex-M processor always executes in Thumb mode so the target address always has the lower bit set. If you disassemble code produced by the compiler, notice that the address it uses for functions is always odd. Also, if you look back in this application note at some of the screen captures showing the exception stack frame, note that the saved value of the LR is also always odd.
5 General Fault Debug Flow

Listed below are the general steps to follow to debug a fault. These are the steps used in the examples earlier in this application note. Often you can collect enough clues to allow you to solve the problem before it becomes necessary to follow all the steps.

1. Examine the NVIC_FAULT_STAT register at 0xE000.ED28 to determine the type of fault.
2. If either the MMARVALID or BFARVALID bits are set, read the corresponding Fault Address Register (MMADR or FAULTADDR) to get the faulting address.
3. Find the value of the stack pointer in the fault handler.
4. If needed, adjust the stack pointer to find the address of the exception stack frame. This step is necessary if your fault handler function allocates storage on the stack. For the StellarisWare examples, no adjustment is needed.
5. Get the value of the saved PC from the exception stack frame.
6. Disassemble the code at the location of the saved PC to find the instruction that caused the fault. Sometimes the saved PC does not point to the exact instruction that caused the fault, but only in the general location.
7. For an imprecise bus fault, look for the most recent store instruction that executed prior to the occurrence of the fault.
8. Use the saved values of the other working registers, as needed, to reconstruct the instruction operands.
9. Use the saved value of the LR to find the prior function in the call stack. Sometimes the saved LR value cannot be used this way.

6 Conclusion

The Stellaris Cortex-M processor provides a powerful fault handling system and several features to help you find the cause of a fault. After debugging a few faults, you will be more comfortable handling and troubleshooting faults as they happen, and will be able to find the cause in a few minutes.

7 References

This application note provides information about Stellaris Cortex-M registers and other features related to fault handling. This information is presented at a summary level and does not serve as a reference for these features. For detailed information about Stellaris fault handling, see the following related documents which are available on the Stellaris web site at www.ti.com/stellaris:

- Stellaris Microcontroller Data Sheet
- Stellaris® Errata
- ARM® Cortex™-M Errata
- The Definitive Guide to the ARM® Cortex-M3 by Joseph Yiu
IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or “enhanced plastic.” Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

<table>
<thead>
<tr>
<th>Products</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audio</td>
<td>Automotive and Transportation</td>
</tr>
<tr>
<td>Amplifiers</td>
<td>Communications and Telecom</td>
</tr>
<tr>
<td>Data Converters</td>
<td>Computers and Peripherals</td>
</tr>
<tr>
<td>DLP® Products</td>
<td>Consumer Electronics</td>
</tr>
<tr>
<td>DSP</td>
<td>Energy and Lighting</td>
</tr>
<tr>
<td>Clocks and Timers</td>
<td>Industrial</td>
</tr>
<tr>
<td>Interface</td>
<td>Medical</td>
</tr>
<tr>
<td>Logic</td>
<td>Security</td>
</tr>
<tr>
<td>Power Mgmt</td>
<td>Space, Avionics and Defense</td>
</tr>
<tr>
<td>Microcontrollers</td>
<td>Video and Imaging</td>
</tr>
<tr>
<td>RFID</td>
<td><a href="http://www.ti.com/audio">www.ti.com/audio</a></td>
</tr>
<tr>
<td>OMAP Mobile Processors</td>
<td><a href="http://www.ti.com/omap">www.ti.com/omap</a></td>
</tr>
<tr>
<td>Wireless Connectiv</td>
<td><a href="http://www.ti.com/wireless-connection">www.ti.com/wireless-connection</a></td>
</tr>
</tbody>
</table>

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2012, Texas Instruments Incorporated