Lecture 7: Exceptions and Interrupts

Hardware cleared Interrupt request

Interrupt Request

Interrupt Pending Status

Handler Mode

Processor Mode

Thread Mode

Thomas Schmid
thomas.schmid@utah.edu

January 31, 2013
Announcements

• Lab 3: IO Attribute Editor Problems
  – From Andrzej K.:
    I noticed another problem with Libero and Lab 3. When the IO Attributes Editor comes up, sometimes the listed port names do not match the ones referred to in the schematic and Lab 1. To fix this problem I went back to "Design Hierarchy " right clicked on "lab3fpga_MSS_0" and selected "set as root". Then Generate Component and Synthesize again. This updated the IO editor with the correct port names.

• uint32_t *reg
  reg = uint32_t* 0x0;
  reg += 1; //This will point to address 0x4 this will change the pointer by size(uint32_t)
  reg[1] = 0; //This will set address 4 to 0 (takes a point and uses the array notation and it will find out the type and then go to that location in memory.
Announcements

• Graduate Student Presentations
  • we have 19 graduate students
  • 15 minute presentation (4 per lecture)
  • Sign-up here:
    - http://goo.gl/HUAXN
• Think about an embedded system you want to present
  • Describe system architecture
  • Where is it used, how does it work, how do I program it
  • Think about why the audience should care
• Examples
  • Quad copters, Imagers, Radios, Electric Cars, Kinect, EnergyMicro Gecko, Cortex-A9, Cortex-R9, FRAM, SmartFusion2, etc.
Announcements

• Saleae Logic Analyzers
  – You have to be a group of 2 to check one out!
Minute Quiz
Interrupts

Merriam-Webster:
- “to break the uniformity or continuity of”

- Informs a program of some external events
- Breaks execution flow

Key questions:
- Where do interrupts come from?
- How do we save state for later continuation?
- How can we ignore interrupts?
- How can we prioritize interrupts?
- How can we share interrupts?
Two key questions to determine how data is transferred to/from a non-trivial I/O device:

1. How does the CPU know when data is available?
   a. Polling
   b. Interrupts

2. How is data transferred into and out of the device?
   a. Programmed I/O
   b. Direct Memory Access (DMA)
Interrupt (a.k.a. exception or trap):
• An event that causes the CPU to stop executing the current program and begin executing a special piece of code called an interrupt handler or interrupt service routine (ISR). Typically, the ISR does some work and then resumes the interrupted program.

Interrupts are really glorified procedure calls, except that they:
• can occur between any two instructions
• are transparent to the running program (usually)
• are not explicitly requested by the program (typically)
• call a procedure at an address determined by the type of interrupt, not the program
Two basic types of interrupts (1/2)

• Those caused by an instruction
  - Examples:
    • TLB miss
    • Illegal/unimplemented instruction
    • div by 0
  - Names:
    • Trap, exception
Two basic types of interrupts (2/2)

• Those caused by the external world
  - External device
  - Reset button
  - Timer expires
  - Power failure
  - System error

• Names:
  - interrupt, external interrupt
How it works

• Something tells the processor core there is an interrupt
• Core transfers control to code that needs to be executed
• Said code “returns” to old program
• Much harder then it looks.
  - Why?
• How do you figure out where to branch to?

• How do you ensure that you can get back to where you started?

• Don’t we have a pipeline? What about partially executed instructions?

• What if we get an interrupt while we are processing our interrupt?

• What if we are in a “critical section?”
• If you know *what* caused the interrupt then you want to jump to the code that handles that interrupt.
  - If you number the possible interrupt cases, and an interrupt comes in, you can just branch to a location, using that number as an offset (this is a branch table)
  - If you don’t have the number, you need to *poll* all possible sources of the interrupt to see who caused it.
    • Then you branch to the right code
• Need to store the return address somewhere.
  - Stack *might* be a scary place.
    • *That* would involve a load/store and might cause an interrupt (page fault)!
  - So a dedicated register seems like a good choice
    • But that might cause problems later…
Snazzy architectures

• A modern processor has *many* (often 50+) instructions in-flight at once.
  - What do we do with them?
• Drain the pipeline?
  - What if one of them causes an exception?
• Punt all that work
  - Slows us down
• What if the instruction that caused the exception was executed before some other instruction?
  - What if that other instruction caused an interrupt?
Nested interrupts

• If we get one interrupt while handling another what to do?
  - Just handle it
    • But what about that dedicated register?
    • What if I’m doing something that can’t be stopped?
  - Ignore it
    • But what if it is important?
  - Prioritize
    • Take those interrupts you care about. Ignore the rest
    • Still have dedicated register problems.
Critical section

• We probably need to ignore some interrupts but take others.
  - Probably should be sure our code can’t cause an exception.
  - Use same prioritization as before.
The Reset Interrupt

1) No power

2) System is held in RESET as long as VCC15 < 0.8V
   a) In reset: registers forced to default
   b) RC-Osc begins to oscillate
   c) MSS_CCC drives RC-Osc/4 into FCLK
   d) PORESET_N is held low

3) Once VCC15GOOD, PORESET_N goes high
   a) MSS reads from eNVM address 0x0 and 0x4
## SmartFusion Interrupts

<table>
<thead>
<tr>
<th>Cortex-M3 NVIC Input</th>
<th>IRQ Label</th>
<th>IRQ Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMI</td>
<td>WDOGTIMEOUT_IRQ</td>
<td>WATCHDOG</td>
</tr>
<tr>
<td>INTISR[60]</td>
<td>WDOGWOAKEUP_IRQ</td>
<td>WATCHDOG</td>
</tr>
<tr>
<td>INTISR[1]</td>
<td>BROWNOUT1_5V_IRQ</td>
<td>VR/PSM</td>
</tr>
<tr>
<td>INTISR[2]</td>
<td>BROWNOUT3_3V_IRQ</td>
<td>VR/PSM</td>
</tr>
<tr>
<td>INTISR[3]</td>
<td>RTCMATCHEVENT_IRQ</td>
<td>RTC</td>
</tr>
<tr>
<td>INTISR[4]</td>
<td>PU_N_IRQ</td>
<td>RTC</td>
</tr>
<tr>
<td>INTISR[5]</td>
<td>EMAC_IRQ</td>
<td>Ethernet MAC</td>
</tr>
<tr>
<td>INTISR[7]</td>
<td>ENVM_0_IRQ</td>
<td>ENVM Controller</td>
</tr>
<tr>
<td>INTISR[8]</td>
<td>ENVM_1_IRQ</td>
<td>ENVM Controller</td>
</tr>
<tr>
<td>INTISR[9]</td>
<td>DMA_IRQ</td>
<td>Peripheral DMA</td>
</tr>
<tr>
<td>INTISR[10]</td>
<td>UART_0_IRQ</td>
<td>UART_0</td>
</tr>
<tr>
<td>INTISR[11]</td>
<td>UART_1_IRQ</td>
<td>UART_1</td>
</tr>
<tr>
<td>INTISR[12]</td>
<td>SPI_0_IRQ</td>
<td>SPI_0</td>
</tr>
<tr>
<td>INTISR[13]</td>
<td>SPI_1_IRQ</td>
<td>SPI_1</td>
</tr>
<tr>
<td>INTISR[14]</td>
<td>I2C_0_IRQ</td>
<td>I2C_0</td>
</tr>
<tr>
<td>INTISR[15]</td>
<td>I2C_0_SMBALERT_IRQ</td>
<td>I2C_0_SMBALERT_IRQ</td>
</tr>
<tr>
<td>INTISR[16]</td>
<td>I2C_0_SMBSUS_IRQ</td>
<td>I2C_0_SMBSUS_IRQ</td>
</tr>
<tr>
<td>INTISR[17]</td>
<td>I2C_1_IRQ</td>
<td>I2C_1</td>
</tr>
<tr>
<td>INTISR[18]</td>
<td>I2C_1_SMBALERT_IRQ</td>
<td>I2C_1_SMBALERT_IRQ</td>
</tr>
<tr>
<td>INTISR[19]</td>
<td>I2C_1_SMBSUS_IRQ</td>
<td>I2C_1_SMBSUS_IRQ</td>
</tr>
<tr>
<td>INTISR[20]</td>
<td>TIMER_1_IRQ</td>
<td>TIMER_1_IRQ</td>
</tr>
<tr>
<td>INTISR[21]</td>
<td>TIMER_2_IRQ</td>
<td>TIMER_2_IRQ</td>
</tr>
<tr>
<td>INTISR[22]</td>
<td>PLLLOCK_IRQ</td>
<td>PLLLOCK_IRQ</td>
</tr>
<tr>
<td>INTISR[23]</td>
<td>PLLLOCKLOS_IRQ</td>
<td>PLLLOCKLOS_IRQ</td>
</tr>
<tr>
<td>INTISR[24]</td>
<td>AMB_ERROR_IRQ</td>
<td>AMB_ERROR_IRQ</td>
</tr>
<tr>
<td>INTISR[25]</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>INTISR[26]</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>INTISR[27]</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>INTISR[28]</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>INTISR[29]</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>INTISR[30]</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>INTISR[31]</td>
<td>FAB_IRQ</td>
<td>FABRIC INTERFACE</td>
</tr>
<tr>
<td>INTISR[32]</td>
<td>GPIO_0_IRQ</td>
<td>GPIO_0_IRQ</td>
</tr>
<tr>
<td>INTISR[33]</td>
<td>GPIO_1_IRQ</td>
<td>GPIO_1_IRQ</td>
</tr>
<tr>
<td>INTISR[34]</td>
<td>GPIO_2_IRQ</td>
<td>GPIO_2_IRQ</td>
</tr>
<tr>
<td>INTISR[35]</td>
<td>GPIO_3_IRQ</td>
<td>GPIO_3_IRQ</td>
</tr>
</tbody>
</table>

### 54 more ACE specific interrupts

GPIO_3_IRQ to GPIO_31_IRQ cut
And the interrupt vectors
(in startup_a2fxxxm3.s found in CMSIS, startup_gcc)

    g_pfnVectors:
         .word  _estack
         .word  Reset_Handler
         .word  NMI_Handler
         .word  HardFault_Handler
         .word  MemManage_Handler
         .word  BusFault_Handler
         .word  UsageFault_Handler
         .word  0
         .word  0
         .word  0
         .word  0
         .word  SVC_Handler
         .word  DebugMon_Handler
         .word  0
         .word  PendSV_Handler
         .word  SysTick_Handler
         .word  WdogWakeup_IRQHandler
         .word  BrownOut_1_5V_IRQHandler
         .word  BrownOut_3_3V_IRQHandler

.................. (they continue)
Pending interrupts

The normal case. Once Interrupt request is seen, processor puts it in “pending” state even if hardware drops the request. IPS is cleared by the hardware once we jump to the ISR.
In this case, the processor never took the interrupt because we cleared the IPS by hand (via a memory-mapped I/O register)
When the processor starts to execute an interrupt, the interrupt becomes active and the pending bit will be cleared automatically (Figure 7.11). When an interrupt is active, you cannot start processing the same interrupt again, until the interrupt service routine is terminated with an interrupt return (also called an exception exit, as discussed in Chapter 9). Then the active status is cleared, and the interrupt can be processed again if the pending status is 1. It is possible to repend an interrupt before the end of the interrupt service routine.

If an interrupt source continues to hold the interrupt request signal active, the interrupt will be pended again at the end of the interrupt service routine as shown in Figure 7.12. This is just like the traditional ARM7TDMI.

If an interrupt is pulsed several times before the processor starts processing it, it will be treated as one single interrupt request as illustrated in Figure 7.13.

If an interrupt is deasserted and then pulsed again during the interrupt service routine, it will be pended again as shown in Figure 7.14.
Pending of an interrupt can happen even if the interrupt is disabled; the pended interrupt can then trigger the interrupt sequence when the enable is set later. As a result, before enabling an interrupt, it could be useful to check whether the pending register has been set. The interrupt source might have been activated previously and have set the pending status. If necessary, you can clear the pending status before you enable an interrupt.

![Diagram showing the flow of interrupt request and status]

**Figure 7.11 Continuous Interrupt Request Pends Again After Interrupt Exit**

**Figure 7.12 Interrupt Pending Only Once, Even with Multiple Pulses Before the Handler**
Pending of an interrupt can happen even if the interrupt is disabled; the pended interrupt can then trigger the interrupt sequence when the enable is set later. As a result, before enabling an interrupt, it could be useful to check whether the pending register has been set. The interrupt source might have been activated previously and have set the pending status. If necessary, you can clear the pending status before you enable an interrupt.

**Figure 7.11** Continuous Interrupt Request Pends Again After Interrupt Exit

**Figure 7.12** Interrupt Pending Only Once, Even with Multiple Pulses Before the Handler
New Interrupt Request after Pending Cleared

Fault Exceptions

A number of system exceptions are useful for fault handling. There are several categories of faults:

- Bus faults
- Memory management faults
- Usage faults
- Hard faults

Bus Faults

Bus faults are produced when an error response is received during a transfer on the AHB interfaces. It can happen at these stages:

- Instruction fetch, commonly called prefetch abort
- Data read/write, commonly called data abort

In the Cortex-M3, bus faults can also occur during a:

- Stack PUSH in the beginning of interrupt processing, called a stacking error
- Stack POP at the end of interrupt processing, called an unstacking error

From: The Definitive Guide to the ARM Cortex-M3
The Nested Vectored Interrupt Controller (NVIC) on the Cortex-M3

- Control registers are memory mapped
- Contains control logic for interrupt processing

- 15 internal interrupts (defined by ARM)
- Supports up to 240 external interrupts (vendor specific)
- Accessed at 0xE000E000 on any Cortex-M3!

- Register definitions can be found at:
  - ARM Cortex-M3 Technical Reference Manual v2.1, Chapter 6
  - The Definitive Guide to the ARM Cortex-M3
# Configuring the NVIC

**Interrupt Set Enable and Clear Enable**

- 0xE000E100-0xE000E11C, 0xE000E180-0xE000E19C

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0xE000E100 | SETENA0    | R/W  | 0     | Enable for external interrupt #0–31
|           |            |      |       | bit[0] for interrupt #0 (exception #16)  |
|           |            |      |       | bit[1] for interrupt #1 (exception #17)  |
|           |            |      |       | ...                                           |
|           |            |      |       | bit[31] for interrupt #31 (exception #47)   |
|           |            |      |       | Write 1 to set bit to 1; write 0 has no effect|
|           |            |      |       | Read value indicates the current status      |

| 0xE000E180 | CLRENA0    | R/W  | 0     | Clear enable for external interrupt #0–31   |
|           |            |      |       | bit[0] for interrupt #0                      |
|           |            |      |       | bit[1] for interrupt #1                      |
|           |            |      |       | ...                                           |
|           |            |      |       | bit[31] for interrupt #31                    |
|           |            |      |       | Write 1 to clear bit to 0; write 0 has no effect|
|           |            |      |       | Read value indicates the current enable status|
### Configuring the NVIC (2)

#### Set Pending & Clear Pending

- 0xE000E200-0xE000E21C, 0xE000E280-0xE000E29C

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000E200</td>
<td>SETPEND0</td>
<td>R/W</td>
<td>0</td>
<td>Pending for external interrupt #0–31 bit[0] for interrupt #0 (exception #16) bit[1] for interrupt #1 (exception #17) ... bit[31] for interrupt #31 (exception #47) Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status</td>
</tr>
<tr>
<td>0xE000E280</td>
<td>CLRPEND0</td>
<td>R/W</td>
<td>0</td>
<td>Clear pending for external interrupt #0–31 bit[0] for interrupt #0 (exception #16) bit[1] for interrupt #1 (exception #17) ... bit[31] for interrupt #31 (exception #47) Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current pending status</td>
</tr>
</tbody>
</table>
The NVIC and Interrupt Control priority-level registers depend on how many external interrupts the chip contains (see Table 8.3). The priority level configuration registers details can be found in Appendix D, Table D.18.

**Table 8.3 Interrupt Priority-Level Registers (0xE000E400-0xE000E4EF)**

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000E400</td>
<td>PRI_0</td>
<td>R/W</td>
<td>0</td>
<td>Priority-level external interrupt #0</td>
</tr>
<tr>
<td>0xE000E401</td>
<td>PRI_1</td>
<td>R/W</td>
<td>0</td>
<td>Priority-level external interrupt #1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xE000E41F</td>
<td>PRI_31</td>
<td>R/W</td>
<td>0</td>
<td>Priority-level external interrupt #31</td>
</tr>
</tbody>
</table>

Active Status

Each external interrupt has an active status bit. When the processor starts the interrupt handler, the bit is set to 1 and cleared when the interrupt return is executed. However, during an interrupt service routine execution, a higher-priority interrupt might occur and cause a preemption. During this period, despite the fact that the processor is executing another interrupt handler, the previous interrupt is still defined as active. The active status registers for external interrupts are read-only (see Table 8.4).

**Table 8.4 Interrupt Active Status Registers (0xE000E300-0xE000E31C)**

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000E300</td>
<td>ACTIVE0</td>
<td>R</td>
<td>0</td>
<td>Active status for external interrupt #0–31</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bit[0] for interrupt #0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bit[1] for interrupt #1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bit[31] for interrupt #31</td>
</tr>
<tr>
<td>0xE000E304</td>
<td>ACTIVE1</td>
<td>R</td>
<td>0</td>
<td>Active status for external interrupt #32–63</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

PRIMASK and FAULTMASK Special Registers

The PRIMASK register is used to disable all exceptions except NMI and hard fault. It effectively changes the current priority level to 0 (highest programmable level). This register is programmable using MRS and MSR instructions. For example:

```
MOV    R0, #1
MSR    PRIMASK, R0   ; Write 1 to PRIMASK to disable all interrupts
```
Interrupt Priority

- What do we do if several interrupts arrive at the same time?
- NVIC allows to set priorities for (almost) every interrupt
- 3 fixed highest priorities, up to 256 programmable priorities
  - 128 preemption levels
  - Not all priorities have to be implemented by a vendor!

- SmartFusion has 32 priority levels, i.e., 0x00, 0x08, …, 0xF8
- Higher priority interrupts can pre-empt lower priorities
- Priority can be sub-divided into priority groups
  - splits priority register into two halves, preempt priority and subpriority
  - preempt priority: indicates if an interrupt can pre-empt another
  - subpriority: used if two interrupts of same group arrive concurrently
Interrupt Priority (2)

- Interrupt Priority Level Registers
  - 0xE000E400-0xE000E4EF

### Table 8.3 Interrupt Priority-Level Registers (0xE000E400-0xE000E4EF)

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000E400</td>
<td>PRI_0</td>
<td>R/W</td>
<td>0 (8-bit)</td>
<td>Priority-level external interrupt #0</td>
</tr>
<tr>
<td>0xE000E401</td>
<td>PRI_1</td>
<td>R/W</td>
<td>0 (8-bit)</td>
<td>Priority-level external interrupt #1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xE000E41F</td>
<td>PRI_31</td>
<td>R/W</td>
<td>0 (8-bit)</td>
<td>Priority-level external interrupt #31</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Active Status

Each external interrupt has an active status bit. When the processor starts the interrupt handler, the bit is set to 1 and cleared when the interrupt return is executed. However, during an interrupt service routine execution, a higher-priority interrupt might occur and cause a preemption. During this period, despite the fact that the processor is executing another interrupt handler, the previous interrupt is still defined as active. The active registers are 32-bit but can also be accessed using half word or byte-size transfers. If there are more than 32 external interrupts, there will be more than one active register. The active status registers for external interrupts are read-only (see Table 8.4).

### Table 8.4 Interrupt Active Status Registers (0xE000E300-0xE000E31C)

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000E300</td>
<td>ACTIVE0</td>
<td>R</td>
<td>Active status for external interrupt #0–31</td>
</tr>
<tr>
<td>0xE000E304</td>
<td>ACTIVE1</td>
<td>R</td>
<td>Active status for external interrupt #32–63</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### PRIMASK and FAULTMASK Special Registers

The PRIMASK register is used to disable all exceptions except NMI and hard fault. It effectively changes the current priority level to 0 (highest programmable level). This register is programmable using MRS and MSR instructions. For example:

```
MOV    R0, #1
MSR    PRIMASK, R0   ; Write 1 to PRIMASK to disable all interrupts
```
Preemption Priority and Subpriority

### Priority Group

<table>
<thead>
<tr>
<th>Priority Group</th>
<th>Preempt Priority Field</th>
<th>Subpriority Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bit [7:1]</td>
<td>Bit [0]</td>
</tr>
<tr>
<td>1</td>
<td>Bit [7:2]</td>
<td>Bit [1:0]</td>
</tr>
<tr>
<td>2</td>
<td>Bit [7:3]</td>
<td>Bit [2:0]</td>
</tr>
<tr>
<td>3</td>
<td>Bit [7:4]</td>
<td>Bit [3:0]</td>
</tr>
<tr>
<td>4</td>
<td>Bit [7:5]</td>
<td>Bit [4:0]</td>
</tr>
<tr>
<td>5</td>
<td>Bit [7:6]</td>
<td>Bit [5:0]</td>
</tr>
<tr>
<td>6</td>
<td>Bit [7]</td>
<td>Bit [6:0]</td>
</tr>
<tr>
<td>7</td>
<td>None</td>
<td>Bit [7:0]</td>
</tr>
</tbody>
</table>

### Application Interrupt and Reset Control Register (Address 0xE000ED0C)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Type</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>VECTKEY</td>
<td>R/W</td>
<td>–</td>
<td>Access key; 0x05FA must be written to this field to write to this register, otherwise the write will be ignored; the read-back value of the upper half word is 0xFA05</td>
</tr>
<tr>
<td>15</td>
<td>ENDIANNESS</td>
<td>R</td>
<td>–</td>
<td>Indicates endianness for data: 1 for big endian (BE8) and 0 for little endian; this can only change after a reset</td>
</tr>
<tr>
<td>10:8</td>
<td>PRIGROUP</td>
<td>R/W</td>
<td>0</td>
<td>Priority group</td>
</tr>
<tr>
<td>2</td>
<td>SYSRESETREQ</td>
<td>W</td>
<td>–</td>
<td>Requests chip control logic to generate a reset</td>
</tr>
<tr>
<td>1</td>
<td>VECTCLRACTIVE</td>
<td>W</td>
<td>–</td>
<td>Clears all active state information for exceptions; typically used in debug or OS to allow system to recover from system error (Reset is safer)</td>
</tr>
<tr>
<td>0</td>
<td>VECTRESET</td>
<td>W</td>
<td>–</td>
<td>Resets the Cortex-M3 processor (except debug logic), but this will not reset circuits outside the processor</td>
</tr>
</tbody>
</table>
Exercise: How many preemption priorities and subpriority levels do we get on the Smart Fusion if we set Priority Group to 5?
• What if we quickly want to disable all interrupts?

• Write 1 into PRIMASK to disable all interrupt except NMI
  – MOV R0, #1
  – MSR PRIMASK, R0

• Write 0 into PRIMASK to enable all interrupts

• FAULTMASK is the same as PRIMASK, but also blocks hard fault (priority -1)

• What if we want to disable all interrupts below a certain priority?

• Write priority into BASEPRI
  – MOV R0, #0x60
  – MSR BASEPRI, R0
What exactly is an interrupt handler?
Vector Table

- Upon an interrupt, the Cortex-M3 needs to know the address of the interrupt handler (function pointer)
- After powerup, vector table is located at 0x00000000

<table>
<thead>
<tr>
<th>Address</th>
<th>Exception Number</th>
<th>Value (Word Size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>–</td>
<td>MSP initial value</td>
</tr>
<tr>
<td>0x00000004</td>
<td>1</td>
<td>Reset vector (program counter initial value)</td>
</tr>
<tr>
<td>0x00000008</td>
<td>2</td>
<td>NMI handler starting address</td>
</tr>
<tr>
<td>0x0000000C</td>
<td>3</td>
<td>Hard fault handler starting address</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>Other handler starting address</td>
</tr>
</tbody>
</table>

- Can be relocated to change interrupt handlers at runtime (vector table offset register)
Vector Table in SoftConsole

• Located in startup_a2fxxxm3.s

```c
15/*---------------------------------------------------------------
16 * Vector table
17 */
18 .global g_pfnVectors
19 .section .isr_vector,"a",@progbits
20 .type g_pfnVectors, @object
21 .size g_pfnVectors, -.g_pfnVectors
22
23 g_pfnVectors:
24 .word _estack
25 .word Reset_Handler
26 .word NMI_Handler
27 .word HardFault_Handler
28 .word MemManage_Handler
29 .word BusFault_Handler
30 .word UsageFault_Handler
31 .word 0
32 .word 0
33 .word 0
34 .word 0
```

• Put at 0x00000000 in linker script

```c
42 SECTIONs
43 {
44 .text :
45 {
46 CREATE_OBJECT_SYMBOLS
47 __text_load = LOADADDR(.text);
48 __text_start = .;
49 *(.isr_vector)
```
Interrupt Handlers

192/*****************************/
193 * Reset_Handler
194 */
195 .global Reset_Handler
196 .type Reset_Handler, %function
197 Reset_Handler:
198 _start:

280/*****************************/
281 * NMI_Handler
282 */
283 .weak NMI_Handler
284 .type NMI_Handler, %function
285 NMI_Handler:
286 B .
287
288/*****************************/
289 * HardFault_Handler
290 */
291 .weak HardFault_Handler
292 .type HardFault_Handler, %function
293 HardFault_Handler:
294 B .
295
Interrupt Handler in GNU C

• We can overwrite the predefined interrupt handlers

```c
__attribute__((__interrupt__)) void Timer1_IRQHandler()
{
    MSS_TIM1_disable_irq();
    MSS_TIM1_clear_irq();
    ...
    NVIC_ClearPendingIRQ(Timer1_IRQn);
}

int main()
{
    MSS_TIM1_enable_irq();
    NVIC_EnableIRQ(Timer1_IRQn);
    ...
    while(1){}
}
```
1. Automatic saving of registers upon exception
   • PC, PSR, R0-R3, R12, LR pushed on the stack
2. While bus busy, fetch exception vector
3. Update SP to new location
4. Update IPSR (low part of PSR) with new exception number
5. Set PC to vector handler
6. Update LR to special value EXC_RETURN

• Several other NVIC registers get updated
• Latency: as short as 12 cycles
The values of PC and PSR are stacked first so that instruction fetch can be started early (which requires modification of PC) and the IPSR can be updated early. After stacking, SP will be updated to N-32 (0/H1100320), and the stacked data arrangement in the stack memory will look like Table 9.1.

<table>
<thead>
<tr>
<th>Address (HADDR)</th>
<th>N-8</th>
<th>N-4</th>
<th>N-32</th>
<th>N-28</th>
<th>N-24</th>
<th>N-20</th>
<th>N-16</th>
<th>N-12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data (HWDATA)</td>
<td>PC</td>
<td>PSR</td>
<td>R0</td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R12</td>
<td>LR</td>
</tr>
</tbody>
</table>

**Figure 9.1 Stacking Sequence**

<table>
<thead>
<tr>
<th>Address (HADDR)</th>
<th>N-8</th>
<th>N-4</th>
<th>N-32</th>
<th>N-28</th>
<th>N-24</th>
<th>N-20</th>
<th>N-16</th>
<th>N-12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data (HWDATA)</td>
<td>PC</td>
<td>PSR</td>
<td>R0</td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R12</td>
<td>LR</td>
</tr>
</tbody>
</table>

The reason the registers R0–R3, R12, LR, PC, and PSR are stacked is that these are caller saved registers, according to C standards (C/C standard Procedure Call Standard for the ARM Architecture, AAPCS, Ref 5). This arrangement allows the interrupt handler to be a normal C function, because registers that could be changed by the exception handler are saved in the stack. The general registers (R0–R3, R12) are located at the end of the stack frame so that they can be easily accessed using SP-related addressing. As a result, it's easy to pass parameters to software interrupts using stacked registers.

**Vector Fetches**

While the data bus is busy stacking the registers, the instruction bus carries out another important task of the interrupt sequence: It fetches the exception vector (the starting address of the exception handler) from the vector table. Since the stacking and vector fetch are performed on separate bus interfaces, they can be carried out at the same time.

You can read the program status registers using the MRS instruction. You can also change the APSR using the MSR instruction, but EPSR and IPSR are read-only. For example:

```
MRS     r0, APSR     ; Read Flag state into R0
MRS     r0, IPSR     ; Read Exception/Interrupt state
MRS     r0, EPSR     ; Read Execution state
MSR     APSR, r0     ; Write Flag state
```

In ARM assembler, when accessing xPSR (all three program status registers as one), the symbol **PSR** is used:

```
MRS     r0, PSR     ; Read the combined program status word
MSR     PSR, r0     ; Write combined program state word
```

The descriptions for the bit fields in PSR are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Bit Description</th>
<th>31 30 29 28 27 26:25 24 23:20 19:16 15:10 9 8 7 6 5 4:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Negative</td>
<td>( )</td>
</tr>
<tr>
<td>Z Zero</td>
<td>( )</td>
</tr>
<tr>
<td>C Carry/borrow</td>
<td>( )</td>
</tr>
<tr>
<td>V Overflow</td>
<td>( )</td>
</tr>
<tr>
<td>Q Sticky saturation flag</td>
<td>( )</td>
</tr>
<tr>
<td>ICI/IT Interrupt-Continuable Instruction (ICI) bits, IF-THEN instruction status bit</td>
<td>( )</td>
</tr>
<tr>
<td>T Thumb state</td>
<td>( )</td>
</tr>
<tr>
<td>Exception Number</td>
<td>( )</td>
</tr>
</tbody>
</table>

**Figure 3.3 Program Status Registers (PSRs) in the Cortex-M3**

<table>
<thead>
<tr>
<th>Bit Description</th>
<th>31 30 29 28 27 26:25 24 23:20 19:16 15:10 9 8 7 6 5 4:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>N Negative</td>
<td>( )</td>
</tr>
<tr>
<td>Z Zero</td>
<td>( )</td>
</tr>
<tr>
<td>C Carry/borrow</td>
<td>( )</td>
</tr>
<tr>
<td>V Overflow</td>
<td>( )</td>
</tr>
<tr>
<td>ICI/IT</td>
<td>( )</td>
</tr>
<tr>
<td>T Thumb state</td>
<td>( )</td>
</tr>
<tr>
<td>Exception Number</td>
<td>( )</td>
</tr>
</tbody>
</table>

**Figure 3.4 Combined Program Status Registers (xPSR) in the Cortex-M3**
Return from ISR

• 3 ways to return from an ISR

<table>
<thead>
<tr>
<th>Return Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BX &lt;reg&gt;</strong></td>
<td>If the EXC_RETURN value is still in LR, we can use the BX LR instruction to perform the interrupt return.</td>
</tr>
<tr>
<td><strong>POP {PC}, or POP {..., PC}</strong></td>
<td>Very often the value of LR is pushed to the stack after entering the exception handler. We can use the POP instruction, either a single POP or multiple POPs, to put the EXC_RETURN value to the program counter. This will cause the processor to perform the interrupt return.</td>
</tr>
<tr>
<td><strong>LDR, or LDM</strong></td>
<td>It is possible to produce an interrupt return using the LDR instruction with PC as the destination register.</td>
</tr>
</tbody>
</table>

• Unstack and reset SP
• Update NVIC registers
Nested Interrupts

• Built into the Cortex-M3 (not every MCU has this)
• Make sure main stack is large enough!

• Two methods:
  – Tail Chaining
  – Late Arrival (preemption)
- If first interrupt has same or higher priority
- Skip stacking/unstacking for efficiency
Late Arrival (Preemption)

- Main stack must be able to hold maximum number of preemptions!

Another feature that improves interrupt performance is late arrival exception handling. When an exception takes place and the processor has started the stacking process, and if during this delay a new exception arrives with higher preemption priority, the late arrival exception will be processed first.

For example, if Exception #1 (lower priority) takes place a few cycles before Exception #2 (higher priority), the processor will behave as shown in Figure 9.3, such that Handler #2 is executed as soon as the stacking completes.

![Figure 9.2 Tail Chaining of Exceptions](image)

![Figure 9.3 Late Arrival Exception Behavior](image)

More on the Exception Return Value

When entering an exception handler, the LR is updated to a special value called EXC_RETURN, with the upper 28 bits all set to 1. This value, when loaded into the PC at the end of the
Different Concepts of Interrupt Sharing

- Number of potential interrupts usually larger than interrupt lines availability on Core
- One peripheral often only has one interrupt
- Different types of events are stored in a status register

Example, UART – IIR, 0x40000008

<table>
<thead>
<tr>
<th>3:0</th>
<th>Interrupt identification bits</th>
<th>R</th>
<th>0b0001</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b110 = Highest priority. Receiver line status interrupt due to overrun error, parity error, framing error or break interrupt. Reading the Line Status Register resets this interrupt.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b0100 = Second priority. Receive data available interrupt modem status interrupt. Reading the Receiver Buffer Register (RBR) or the FIFO drops below the trigger level resets this interrupt.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b1100 = Second priority. Character timeout indication interrupt occurs when no characters have been read from the RX FIFO during the last four character times and there was at least one character in it during this time. Reading the Receiver Buffer Register (RBR) resets this interrupt.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b0010 = Third priority. Transmitter Holding Register Empty interrupt. Reading the IIR or writing to the Transmitter Holding Register (THR) resets the interrupt.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0b0000 = Fourth priority. Modem status interrupt due to Clear to Send, Data Set Ready, Ring Indicator, or Data Carrier Detect being asserted. Reading the Modem Status Register resets this interrupt.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This register is read only; writing has no effect. Also see Table 15-9.
ISR Sharing, i.e., Callbacks in C

- There is only one interrupt handler
- Functions have to “subscribe” for events
- Callbacks
  - Driver provides function to register a function pointer
  - Driver stores function pointers in list
  - Upon interrupt, each registered function gets called

```c
typedef void (*radioalarm_handler_t)(void);
radioalarm_handler_t radio_alarm_fired;

void RadioAlarm_init(radioalarm_handler_t handler)
{
    radio_alarm_fired = handler;
}

__attribute__((__interrupt__)) void Timer1_IRQHandler()
{
    alarm_state = FREE;
    MSS_TIM1_disable_irq();
    MSS_TIM1_clear_irq();
    NVIC_ClearPendingIRQ(Timer1_IRQn);
    (*(radio_alarm_fired))(); // call the callback function
}
```
Common Problems and Pit-Falls

- Too many interrupts
  - Your core can’t keep up with handling interrupts

- Concurrency issues
  - One interrupt handler modifies global variables
  - Can be avoided using atomic sections protected through PRIMASK

- Lost interrupts
  - It can happen that an interrupt doesn’t get treated by the Core
  - State machine and peripheral has to be aware of this possibility
  - Danger for deadlocks
Summary

• Overwrite default Interrupt Handler

• Initialization
  – Enable interrupt in NVIC
  – Enable interrupt in Peripheral

• Upon Interrupt
  – Clear interrupt in Peripheral
  – Clear pending bit in NVIC
  – *Potentially disable interrupts temporarily*
Questions?

Comments?

Discussion?

- One point that really **sticks out** about this lecture
- One point that was **unclear** about this lecture