• How do you figure out *where* to branch to?
  *With interrupt number, use interrupt branch table*
• How do you ensure that you can get back to where you started?
  *Store return address in Link Register*
• Don’t we have a pipeline? What about partially executed instructions?
  *Flush the pipeline*
• What if we get an interrupt while we are processing our interrupt?
  *Ignore, prioritize, handle*
• What if we are in a “critical section?”
  *Prioritize and prevent exceptions*
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.
Interrupt Request not Cleared

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]

**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**

From: The Definitive Guide to the ARM Cortex-M3
Multiple Interrupt Pulses

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 of Interrupt Pulses](image)

**Multiple interrupt pulses before entering ISR**

**Interrupt Request**

**Interrupt Pending Status**

**Interrupt Active Status**

**Processor Mode**

![Figure 7.11 Continuous Interrupt Request Pends Again After Interrupt Exit](image)

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

**Interrupt Request stays active**

**Interrupt re-entered**

**Interrupt returned**

![Figure 7.12 Interrupt Pending Only Once, Even with Multiple Pulses Before the Handler](image)

**Interrupt returned**

**Figure 7.12 Interrupt Pending Only Once, Even with Multiple Pulses Before the Handler**

From: The Definitive Guide to the ARM Cortex-M3
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

Interrupt Request after Pending Cleared

Interrupt request pulsed again

Figure 7.13 Interrupt Pending Occurs Again During the Handler

From: The Definitive Guide to the ARM Cortex-M3
Late Arrival

Interrupt #1 (Low Priority)
Interrupt #2 (High Priority)

Processor State

Thread | Exception Sequence

Data Bus

Instruction Bus

Stacking
### 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>Reset 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 |
### Set Pending & Clear Pending

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Reset Value</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</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bit[0] for interrupt #0 (exception #16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bit[1] for interrupt #1 (exception #17)</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 (exception #47)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write 1 to set bit to 1; write 0 has no effect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bit[0] for interrupt #0 (exception #16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bit[1] for interrupt #1 (exception #17)</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 (exception #47)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write 1 to clear bit to 0; write 0 has no effect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Read value indicates the current pending status</td>
</tr>
</tbody>
</table>
LDR     R0  =0xE000E100
MOVS    R1  #0x1
STR     R1  [R0]

Table 8.1  Interrupt Set Enable Registers and Interrupt Clear Enable Registers
(0xE000E100-0xE000E11C, 0xE000E180-0xE000E19C)

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000E100</td>
<td>SETENA0</td>
<td>R/W</td>
<td>0</td>
<td>Enable for external Interrupt #0–31 bit[0] for Interrupt #0 (exception #16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bit[1] for Interrupt #1 (exception #17)</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 (exception #47)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write 1 to set bit to 1; write 0 has no effect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Read value indicates the current status</td>
</tr>
<tr>
<td>0xE000E104</td>
<td>SETENA1</td>
<td>R/W</td>
<td>0</td>
<td>Enable for external Interrupt #32–63</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write 1 to set bit to 1; write 0 has no effect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Read value indicates the current status</td>
</tr>
<tr>
<td>0xE000E108</td>
<td>SETENA2</td>
<td>R/W</td>
<td>0</td>
<td>Enable for external Interrupt #64–95</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write 1 to set bit to 1; write 0 has no effect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Read value indicates the current status</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
<tr>
<td>0xE000E180</td>
<td>CLRENA0</td>
<td>R/W</td>
<td>0</td>
<td>Clear enable 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>…</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?

-3  -2  -1
Reset  NMI  Hard Fault

Programmable Exceptions

Preempt levels with priority group set to 5

Subpriority levels

0x00  0x08  0x10  0x18  0x20  0x28  0x30  0x38  0x40  0x48

Bit 7  Bit 6  Bit 5  Bit 4  Bit 3
Preempt  Sub

0x00  0x40  0x80  0xc0

0x00  0x08  0x10  0x18  0x20  0x28  0x30  0x38  0x40  0x48

Bit 2  Bit 1  Bit 0

0xc0  0xc8  0xd0  0xd8  0xe0  0xe8  0xf0  0xf8
Disable Interrupts

- Disable all interrupts except NMI
  - MOV R0, #1
  - MSR PRIMASK, R0

- FAULTMASK but also blocks hard fault
  - MOV R0, #1
  - MSR FAULTMASK, R0

- What if we want to disable all interrupts below a certain priority?
  - MOV R0, #0xC0
  - MSR BASEPRI, R0
Upon an interrupt, the Cortex-M3 needs to know the address of the interrupt handler (function pointer).

After powerup, the vector table is located at 0x00000000.

### Vector Table

<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).
Interrupt Handlers

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

280;/*-----------------------------
281 * NMI_Handler
282 */
283 .weak  NMI_Handler
284 .type  NMI_Handler, %function
285NMI_Handler:
286   B .
287
288;/*-----------------------------
289 * HardFault_Handler
290 */
291 .weak  HardFault_Handler
292 .type  HardFault_Handler, %function
293HardFault_Handler:
294   B .
295```
• We can overwrite the predefined interrupt handlers

```c
__attribute__((__interrupt__)) void Timer1_IRQHandler()
{
...
}

int main()
{
...
    while(1){}
}
```
### 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>Bit</th>
<th>Interrupt Identification Bits</th>
<th>R</th>
<th>0b0001</th>
</tr>
</thead>
<tbody>
<tr>
<td>7:6</td>
<td>Mode</td>
<td>0b11</td>
<td>Always</td>
</tr>
<tr>
<td>5:4</td>
<td>Reserved</td>
<td>0b0</td>
<td>Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.</td>
</tr>
<tr>
<td>3:0</td>
<td>Interrupt identification bits</td>
<td>0b0110 = 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. 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. 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 Receive Buffer Register (RBR) resets this interrupt. 0b0010 = Third priority. Transmitter Holding Register Empty interrupt. Reading the IIR or writing to the Transmit Holding Register (THR) resets the interrupt. 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. This register is read only; writing has no effect. Also see Table 15-9.</td>
<td></td>
</tr>
</tbody>
</table>
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
}
```