Recap of the last lecture

- Why is Reset Vector +1?
  - It's an ARM specific thing. The least significant bit in jump instructions indicates the type of instruction at that location (0: for ARM, 1: for Thumb). Since the Cortex-M3 can only execute Thumb2, this will always...
This will provide a saturation feature that has the properties shown in Figure 4.4.

Saturation for a signed value into an unsigned value

\[ \text{SSAT.W } <Rd>, \#<\text{immed}>, <Rn>, \{,<\text{shift}>\} \]
\[ \text{SSAT.W } R1, \#16, R0 \]

<table>
<thead>
<tr>
<th>Input (R0)</th>
<th>Output (R1)</th>
<th>Q Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00020000</td>
<td>0x00007FFF</td>
<td>Set</td>
</tr>
<tr>
<td>0x00008000</td>
<td>0x00007FFF</td>
<td>Set</td>
</tr>
<tr>
<td>0x00007FFF</td>
<td>0x00007FFF</td>
<td>Unchanged</td>
</tr>
<tr>
<td>0x00000000</td>
<td>0x00000000</td>
<td>Unchanged</td>
</tr>
<tr>
<td>0xFFFF8000</td>
<td>0xFFFF8000</td>
<td>Unchanged</td>
</tr>
<tr>
<td>0xFFFF8001</td>
<td>0xFFFF8000</td>
<td>Set</td>
</tr>
<tr>
<td>0xFFFF8001</td>
<td>0xFFFF8000</td>
<td>Set</td>
</tr>
<tr>
<td>0xFFFFE000</td>
<td>0xFFFF8000</td>
<td>Set</td>
</tr>
</tbody>
</table>

From: The Definitive Guide to the ARM Cortex-M3

### Generalization of Interrupts

- Merriam-Webster: \textit{“to break the uniformity or continuity of”}
- Informs a program of some external events
- Breaks execution flow
- 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?
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 FSCK
   d. PORESET_N is held low
3. Once VCC15GOOD, PORESET_N goes high
   a. MSS reads from eNVM address 0x0 and 0x4

Interrupt Handling

Source → Controller → MPU

• On the Cortex-M3
  - Source: Software, Peripheral
  - Controller: Nested Vectored Interrupt Controller (NVIC)
  - MPU: Cortex-M3 Core

The Reset Interrupt (2)

• The Reset Interrupt is Non-Maskable!

Sources of Interrupts
Types of Interrupts

- Physical interrupts
  - Level-triggered
  - Edge-triggered (positive, negative)
  - Hybrid
    - Look for edges, but signal must stay for a while
    - Often used for non-maskable interrupts to avoid glitches
- Non-maskable interrupts
- Interrupt priorities
- Software interrupts

The Nested Vectored Interrupt Controller (NVIC) on the Cortex-M3

- Control registers are memory mapped
- Contains control logic for interrupt processing
- Also contains MPU, SYSTICK Timer, and Debug

- 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

Actel SmartFusion Interrupts

- GPIO_31_IRQ to GPIO_31_IRQ cut

54 more ACE specific interrupts
When the booting process is done, you can define a part of your SRAM as the new vector process. Other exceptions cannot take place until they are enabled. These are required because the NMI and hard fault can potentially occur during your boot with an interrupt return (also called an exception exit). You cannot start processing the same interrupt again until the interrupt service routine is terminated. The pending bit will be cleared automatically (Figure 7.10). When an interrupt is active, you cannot clear a pending interrupt or use software to pend a new interrupt by setting the pending register.

When the processor starts to execute an interrupt, the interrupt becomes active and the interrupt request line is asserted. The corresponding interrupt service routine (ISR) is then triggered. The interrupt can be accessed in the NVIC and is writable, so you can clear a pending interrupt or re-pend an interrupt. When an interrupt input is asserted, it will be pended. Even if the interrupt source de-asserts it, the pended interrupt status will still cause the interrupt handler to be executed when the priority is allowed. However, if the pending status is cleared before the processor starts responding to the interrupt, the pended interrupt status will still cause the interrupt handler to be executed.

If an interrupt is de-asserted and then pulsed again during the interrupt service routine, it will be treated 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 7.12.

If an interrupt source continues to hold the interrupt request signal active, the interrupt will be active status set during handler execution. Active Status set during handler execution is cleared by software. If an interrupt request stays active, it could be useful to check whether the pending register has been set. The interrupt then trigger the interrupt sequence when the enable is set later. As a result, before enabling an interrupt, you can clear the pending status before you enable an interrupt. If the interrupt is pending status cleared by software, it could be useful to check whether the pending register has been set. The interrupt then trigger the interrupt sequence when the enable is set later. As a result, before enabling an interrupt, you can clear the pending status before you enable an interrupt.
can clear the pending status before you enable an interrupt. If an interrupt is pending and you want to ensure that it is processed, it could be useful to check whether the pending register has been set. The interrupt then triggers the interrupt sequence when the enable is set later. As a result, before enabling an interrupt, you can check if the pending status register has been set.

From: 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>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 |

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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>
</table>
| 0xE000E200 | SETPENO | R/W  | 0           | 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 |

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0xE000E280 | CLRPNOD | R/W  | 0           | 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 |
Configuring the NVIC (3)

- Interrupt Active Status Register
  - 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>

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!

<table>
<thead>
<tr>
<th>Bit</th>
<th>Implemented</th>
<th>Not implemented, read as zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 preempt another
  - subpriority: used if two interrupts of same group arrive concurrently

Interrupt Priority (2)

- 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>

Preemption Priority and Subpriority

<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>Read/Write</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>VECTKEY</td>
<td>R/W</td>
<td></td>
<td>Access bit, must be written to this field to enable or disable interrupts;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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; 1 for big endian (BE) and 0 for little endian; this</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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>SYRESETREQ</td>
<td>W</td>
<td></td>
<td>Requests chip control logic to generate a reset</td>
</tr>
<tr>
<td>1</td>
<td>VECTCASSERT</td>
<td>W</td>
<td></td>
<td>Clears all active state information for exceptions, typically used in debug</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>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?

-3  -2  -1  0x00
<table>
<thead>
<tr>
<th></th>
<th>Preempt levels with priority group set to 5</th>
<th>Subpriority levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NMI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hard Fault</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Programmable Exceptions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PRIMASK, FAULTMASK, and BASEPRI

- 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>0</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
- Put at 0x00000000 in linker script

Interrupt Handlers

```c
#include <stdio.h>

#define MAIN_STACK_SIZE 0x1000

#define __attribute__ ((__interrupt__))

#define Timer1_IRQHandler
{
  MSS_TIM1_disable_irq();
  MSS_TIM1_clear_irq();
  NVIC_ClearPendingIRQ(Timer1_IRQn);
}

#define int main()
{
  MSS_TIM1_enable_irq();
  NVIC_EnableIRQ(Timer1_IRQn);
  ...
  while(1){}
}
```

Interrupt Service Routines

1. Automatic saving of registers upon exception
   - PC, PSR, R0-R3, R12, LR pushed on the stack
2. While 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
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>BX &lt;reg&gt;</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>POP (PC), or POP (...; PC)</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>LDR, or LDM</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)

Tail Chaining

- 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!
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

Interrupt identification bits  R  0x0000
0b1110 = 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.
0b0101 = Second priority. Receive data available interrupt. Reading the Receiver Buffer Register (RBBR) or the FIFO drops below the trigger level resets this interrupt.
0b0110 = Second priority. Receive data available interrupt. Reading the Receiver Buffer Register (RBBR) or the FIFO drops below the trigger level resets this interrupt.
0b0111 = Second priority. Receive data available interrupt. Reading the Receiver Buffer Register (RBBR) or the FIFO drops below the trigger level resets this interrupt.
0b1000 = Fourth priority. Receiver data available interrupt. Reading the Receiver Buffer Register (RBBR) or the FIFO drops below the trigger level resets this interrupt.
0b1001 = Fourth priority. Transmit Holding Register Empty interrupt. Reading the Transmit Holding Register (THR) resets the interrupt.
0b1010 = Third priority. Transmit Holding Register Empty interrupt. Reading the Transmit Holding Register (THR) resets the interrupt.
0b1011 = Third priority. Transmit Holding Register Empty interrupt. Reading the Transmit Holding Register (THR) resets the interrupt.
0b1100 = Second priority. Modem status interrupt. Reading the Modem Status Register resets this interrupt.
0b1101 = Second priority. Modem status interrupt. Reading the Modem Status Register resets this interrupt.
0b1110 = Highest priority. Modem status interrupt. Reading the Modem Status Register resets this interrupt.

Different Concepts of Interrupt Sharing

- 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