High-level review of interrupts

- Why do we need them? Why are the alternatives unacceptable?
  - Convince me!
- What sources of interrupts are there?
  - Hardware and software!
- What makes them difficult to deal with?
  - Interrupt controllers are complex: there is a lot to do!
    - Enable/disable, prioritize, allow preemption (nested interrupts), etc.
  - Software issues are non-trivial
    - Can’t trash work of task you interrupted
    - Need to be able to restore state
    - Shared data issues are a real pain

### Table 7.1: List of External Exceptions

<table>
<thead>
<tr>
<th>Exception Number</th>
<th>Exception Type</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>Reset</td>
<td>0</td>
<td>Non-maskable interrupt (verified monitor input)</td>
</tr>
<tr>
<td>0x00000001</td>
<td>First Fault</td>
<td>1</td>
<td>Memory management fault (not enabled)</td>
</tr>
<tr>
<td>0x00000002</td>
<td>Second Fault</td>
<td>2</td>
<td>Memory management fault (not enabled)</td>
</tr>
<tr>
<td>0x00000003</td>
<td>Bus Fault</td>
<td>3</td>
<td>Bus fault (not enabled)</td>
</tr>
<tr>
<td>0x00000004</td>
<td>Usage Fault</td>
<td>4</td>
<td>Usage fault (not enabled)</td>
</tr>
</tbody>
</table>

### Table 7.2: List of External Interrupts

<table>
<thead>
<tr>
<th>Exception Number</th>
<th>Exception Type</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>External Interrupt 0</td>
<td>Programmable</td>
<td>External interrupt</td>
</tr>
<tr>
<td>0x00000001</td>
<td>External Interrupt 1</td>
<td>Programmable</td>
<td>External interrupt</td>
</tr>
<tr>
<td>0x00000002</td>
<td>External Interrupt 2</td>
<td>Programmable</td>
<td>External interrupt</td>
</tr>
</tbody>
</table>

### Configuring the NVIC

#### Interrupt Set Enable and Clear Enable

- **0x00000000** SETENA0
- **0x00000001** CLEARA0

#### Table 7.1: List of External Exceptions

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>ACTVEH1</td>
<td>R</td>
<td>0</td>
<td>Active status for internal interrupt #0</td>
</tr>
</tbody>
</table>
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.

This figure and those following are from "The Definitive Guide to the ARM Cortex-M3, Section 7.4"

New Interrupt Request after Pending Cleared

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, preemption priority and subpriority
  - preemption 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

Preemption Priority and Subpriority

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
Interrupt Service Routines

- Automatic saving of registers upon exception
  - PC, PSR, R0-R3, R12, LR
  - This occurs over data bus
- While data bus busy, fetch exception vector
  - i.e. target address of exception handler
  - This occurs over instruction bus
- Update SP to new location
- Update IPSR (low part of xPSR) with exception new #
- Set PC to vector handler
- Update LR to special value EXC_RETURN
- Several other NVIC registers gets updated
- Latency can be as short as 12 cycles (w/o mem delays)

The xPSR register layout

The APSR, IPSR and EPSR registers are allocated to mutually-exclusive bits within a 32-bit register. The combination of the APSR, IPSR and EPSR registers is referred to as the xPSR register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APSR</td>
<td>Interrupt Status and Programme Status Register</td>
</tr>
<tr>
<td>IPSR</td>
<td>Instruction Set Status Register</td>
</tr>
<tr>
<td>EPSR</td>
<td>Exception Status Register</td>
</tr>
</tbody>
</table>

ARM interrupt summary

1. We’ve got a bunch of memory-mapped registers that control things (NVIC)
   - Enable/disable individual interrupts
   - Set/clear pending
   - Interrupt priority and preemption

2. We’ve got to understand how the hardware interrupt lines interact with the NVIC

3. And how we figure out where to set the PC to point to for a given interrupt source.

1. NVIC registers (example)

- 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>SETPEN0</td>
<td>R/W</td>
<td>0</td>
<td>Priority-level interrupt #0</td>
</tr>
<tr>
<td>0xE000E204</td>
<td>PR0_0</td>
<td>R/W</td>
<td>0 (0-bit)</td>
<td>Priority-level interrupt #0</td>
</tr>
<tr>
<td>0xE000E208</td>
<td>PR0_1</td>
<td>R/W</td>
<td>0 (0-bit)</td>
<td>Priority-level interrupt #1</td>
</tr>
<tr>
<td>0xE000E21C</td>
<td>PR0_31</td>
<td>R/W</td>
<td>0 (0-bit)</td>
<td>Priority-level interrupt #31</td>
</tr>
</tbody>
</table>

1. More registers (example)

- Interrupt Priority Level Registers
  - 0xE000E400-0xE000E41F

<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>PR0_0</td>
<td>R/W</td>
<td>0 (0-bit)</td>
<td>Priority-level interrupt #0</td>
</tr>
<tr>
<td>0xE000E404</td>
<td>PR0_1</td>
<td>R/W</td>
<td>0 (0-bit)</td>
<td>Priority-level interrupt #1</td>
</tr>
<tr>
<td>0xE000E41C</td>
<td>PR0_31</td>
<td>R/W</td>
<td>0 (0-bit)</td>
<td>Priority-level interrupt #31</td>
</tr>
</tbody>
</table>
1. Yet another part of the NVIC registers!

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

Application Interrupt and Reset Control Register (Address 0xED00ED00)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Type</th>
<th>Reset Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-16</td>
<td>VECTAddr</td>
<td>R/W</td>
<td>-</td>
<td>Access key. DSPR must be written to this field to write to the register, otherwise the write will be ignored. This read back value of the upper 16 bits is [0x00]</td>
</tr>
<tr>
<td>15-8</td>
<td>EN.removeEventListener</td>
<td>R</td>
<td>-</td>
<td>Indicates readiness for data. T for big endianness (BE) and F for little endianness, this can only change after a reset.</td>
</tr>
<tr>
<td>7-0</td>
<td>PRECED</td>
<td>R/W</td>
<td>0</td>
<td>Priority group</td>
</tr>
<tr>
<td>15</td>
<td>VECTCLEN</td>
<td>W</td>
<td>-</td>
<td>Cleared at reset state for exceptions; typically used in adding or sub to allow entry to recover from errors ever smaller scale.</td>
</tr>
<tr>
<td>8</td>
<td>VECTRSR</td>
<td>W</td>
<td>-</td>
<td>Requests drop control to generate a reset.</td>
</tr>
</tbody>
</table>

2. How external lines interact with the NVIC

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.

This figure and those following are from The Definitive Guide to the ARM Cortex-M3, Section 7.4

3. How the hardware figures out what to set the PC to

```assembly
    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  0  
        .word  0  
        .word  SVCS_Handler  
        .word  SysTick_Handler  
        .word  0  
        .word  PendSV_Handler  
        .word  WdogWakeup_IRQHandler  
        .word  BrownOut_1_5V_IRQHandler  
        .word  BrownOut_3_3V_IRQHandler  
        ................ (they continue)  
```

Discussion: So let’s say a GPIO pin goes high
- When will we get an interrupt?
- What happens if the interrupt is allowed to proceed?

What happens when we return from an ISR?

- **Interrupt exiting process**
  - System restoration needed (different from branch)
  - Special LR value could be stored (0xFFFFFFFFx)

- **Tail chaining**
  - When new exception occurs
  - But CPU handling another exception of same/higher priority
  - New exception will enter pending state
  - But will be executed before register unstacking
  - Saving unnecessary unstacking/stacking operations
  - Can reenter handler in as little as 6 cycles

- **Late arrivals (ok, so this is actually on entry)**
  - When one exception occurs and stacking commences
  - Then another exception occurs before stacking completes
  - And second exception of higher preempt priority arrives
  - The later exception will be processed first

Other stuff: The xPSR register layout

<table>
<thead>
<tr>
<th>APRN</th>
<th>Z</th>
<th>V</th>
<th>C</th>
<th>N</th>
<th>E</th>
<th>XSTT</th>
<th>ISTT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The APRN, IPSR, and EPSR registers are allocated as mutually exclusive bitfields within a 32-bit register. The combination of the APRN, IPSR, and EPSR registers is referred to as the xPSR register.
Example of Complexity: 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

Interrupt types

- Two main types
  - Level-triggered
  - Edge-triggered

Level-triggered interrupts

- Signaled by asserting a line low or high
- Interrupting device drives line low or high and holds it there until it is serviced
- Device deasserts when directed to or after serviced
- Can share the line among multiple devices (w/ OD+PU)
- Active devices assert the line
- Inactive devices let the line float
- Easy to share line w/o losing interrupts
- But servicing increases CPU load → example
- And requires CPU to keep cycling through to check
- Different ISR costs suggests careful ordering of ISR checks
- Can’t detect a new interrupt when one is already asserted

Edge-triggered interrupts

- Signaled by a level "transition" (e.g. rising/falling edge)
- Interrupting device drive a pulse (train) onto INT line
- What if the pulse is too short? Need a pulse extender!
- Sharing "is" possible...under some circumstances
- INT line has a pull up and all devices are OC/OD.
- Devices "pulse" lines
- Could we miss an interrupt? Maybe...if close in time
- What happens if interrupts merge? Need one more ISR pass
- Must check trailing edge of interrupt
- Easy to detect "new interrupts"
- Benefits: more immune to unserviceable interrupts
- Pitfalls: spurious edges, missed edges
- Source of "lockups" in early computers