EECS 373

Design of Microprocessor-Based Systems

Ron Dreslinski
University of Michigan

Lecture 6 & 7: Interrupts (end of slides on Wednesday)

September 26th & 28th

Exceptions, Traps, Faults & ARM's Nested Vectored Interrupt Controller

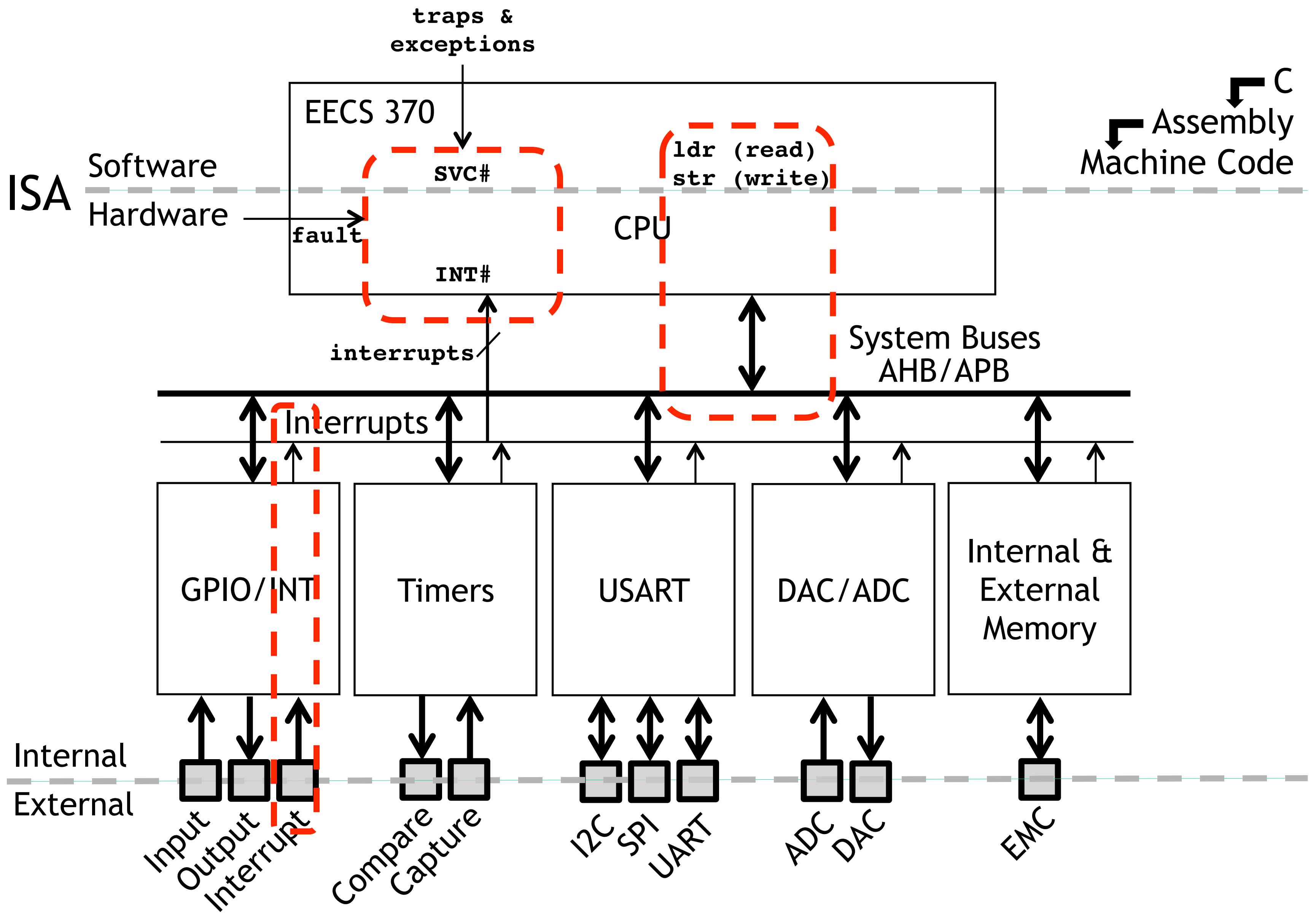
Slides developed in part by Prabal Dutta and Mark Brehob

Administrative

- HW2 Due Today
- Prof. Dreslinski Office hours changed this week, they will be:
  – Tuesday 2:30-3:30pm
  – Wednesday 3-4pm

- Fire Alarm Testing on Wednesday @1:30
  – WE WILL HAVE CLASS

This Week: Interrupts, exceptions, traps, faults, & NVIC



Dealing with asynchronous events

- Many sources of "events" during program execution
  - Application makes a system call
  - Software executes instruction illegally (e.g. divides by zero)
  - Peripheral needs attention or has completed a requested action
- How do we know that an event has occurred?
- Broadly, two options to "detect" events
  - Polling
    - We can repeatedly poll the app/processor/peripherals
    - When an event occurs, detect this via a poll and take action
  - Interrupts
    - Let the app/processors/peripheral notify us instead
    - Take action when such a notification occurs (or shortly later)

Polling-Driven Application

- Recall pushbutton-LED example

```
loop:   mov   r0, #0x4       % PBS MMIO address
        mov   r1, #0x5       % LED MMIO address
        ldr   r2, [r0, #0]   % Read value from switch [1 cycle]
        str   r2 [r1, #0]    % Save value to LED [1 cycle]
        b     loop           % Repeat these steps [1 cycle]
```

- This is a polling-driven application
- Software constantly loops, polling and (re)acting
- However, it doesn’t do anything else useful!

The Problem with Polling

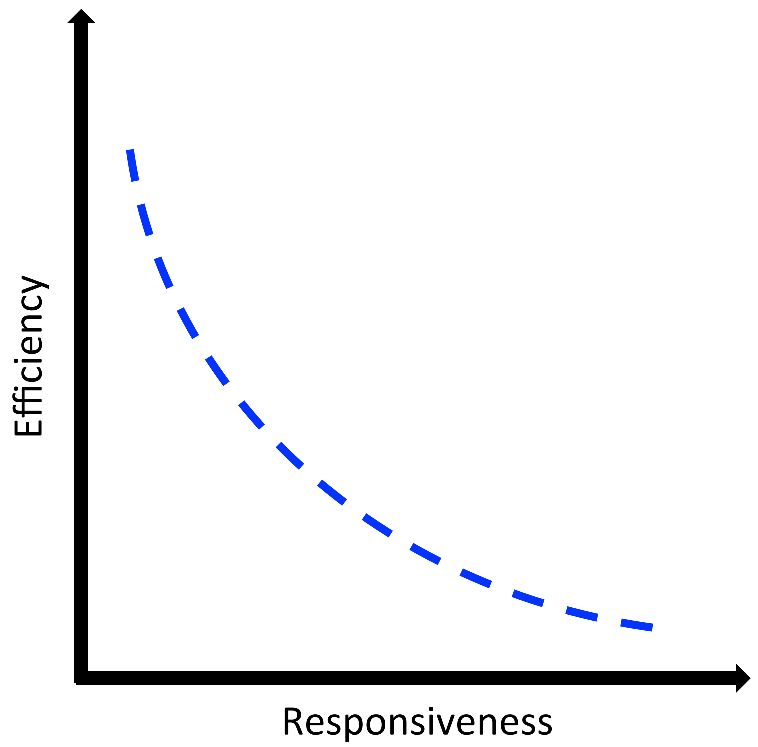
- If we want to do other work, we might call a routine:

```
        mov     r0, #0x4          % PBS MMIO address
        mov     r1, #0x5          % LED MMIO address
loop:   ldr     r2, [r0, #0]      % Read value from switch [1 cycle]
        str     r2 [r1, #0]       % Save value to LED [1 cycle]
        bl      do_some_work      % Do some other work [100 cycles]
        b       loop              % Repeat these steps [1 cycle]
```

- Polling affects the responsiveness of PBS ⇔ LED path!
  - Whenever we're "doing some work," we not polling PBS
  - And the more "other work" we do, the worse the latency gets
- And it affects the efficiency of the processor
  - The ldr/str values don't change very either much
  - So, the processor is mostly wasting CPU cycles (and energy)

Polling trades off efficiency and responsiveness

```
        mov     r0, #0x4        % PBS MMIO address
        mov     r1, #0x5        % LED MMIO address
loop:   ldr     r2, [r0, #0]    % Read value from switch [1 cycle]
        str     r2 [r1, #0]     % Save value to LED [1 cycle]
        bl      do_some_work    % Do some other work [100 cycles]
        b       loop            % Repeat these steps [1 cycle]
```



- Efficiency
  - Minimizing useless work
  - Maximizing useful work
  - Saving cycles & energy
- Responsiveness
  - Minimizing latency
  - Tight event-action coupling
- Can we do better? Yes!

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?

Interrupts

Interrupt (a.k.a. exception or trap):

- An event that causes the CPU to stop executing current program
- Begin executing a special piece of code
  - Called an **interrupt handler** or **interrupt service routine** (ISR)
    - Typically, the ISR does some work
    - 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
    - SVC (supervisor call, e.g.: SVC #3)
  - 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

External interrupt types

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

Level-triggered interrupts

- Basics:
  - Signaled by asserting a line low or high
  - Interrupting device drives line low or high and <u>*holds it there until it is serviced*</u>
  - Device deasserts when directed to or after serviced
    - Requires some way to tell it to stop.

- Sharing?
  - Can share the line among multiple devices
  - Often open-collector or HiZ
    - Active devices assert the line, inactive devices let the line float
  - Easy to share line w/o losing interrupts
  - But servicing increases CPU load
  - 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

- Basics:
  - Signaled by a level *transition* (e.g. rising/falling edge)
  - Interrupting device drives a pulse onto INT line
- Sharing *is* possible
  - INT line has a pull up and all devices are OC/OD.
  - Could we miss an interrupt? Maybe...if close in time
  - What happens if interrupts merge? Need one more ISR pass
  - Easy to detect "new interrupts”
  - Pitfalls: spurious edges, missed edges
- Source of "lockups" in early computers

Exercise: Another case where polling is slow—sharing!

- Assume you have
  - *n* possible interrupt sources
  - That all share a single interrupt line and/or handler
  - That all fire at about the same rate on average
  - And that require about the same amount of time to poll
- The handler might look something like this

```
isr_handler:    bl      chk_interrupt_src_1     % 100 cycles
                bl      chk_interrupt_src_2     % 100 cycles
                …
                bl      chk_interrupt_src_n     % 100 cycles
                bx      lr
```

- How does average interrupt processing time grow with *n*?
- How would you order chk_interrupt_src if the interrupts fired at different rates or had different polling times?

Why are interrupts useful? Example: I/O Data Transfer

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 are available?
   a. Polling
   b. Interrupts

2. How are data transferred into and out of the device?
   a. Programmed I/O
   b. Direct Memory Access (DMA)

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?

Devil is in the details

- How do you figure out *where* to branch to?
- How to you ensure that you can get back to where you started?
- Don’t we have a pipeline? What about partially executed instructions (and OoO instructions)?
- What if we get an interrupt while we are processing an interrupt?
- What if we are in a “critical section?”

Where

- 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

Get back to where you once belonged

- 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…
    - What happens if another interrupt happens?
      - Could that overwrite the register?

Modern 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.
- What about instructions that shouldn’t be interrupted?
  - Disable interrupts while processing an interrupt?

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 premption (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

Our processor—ARM Cortex-M3

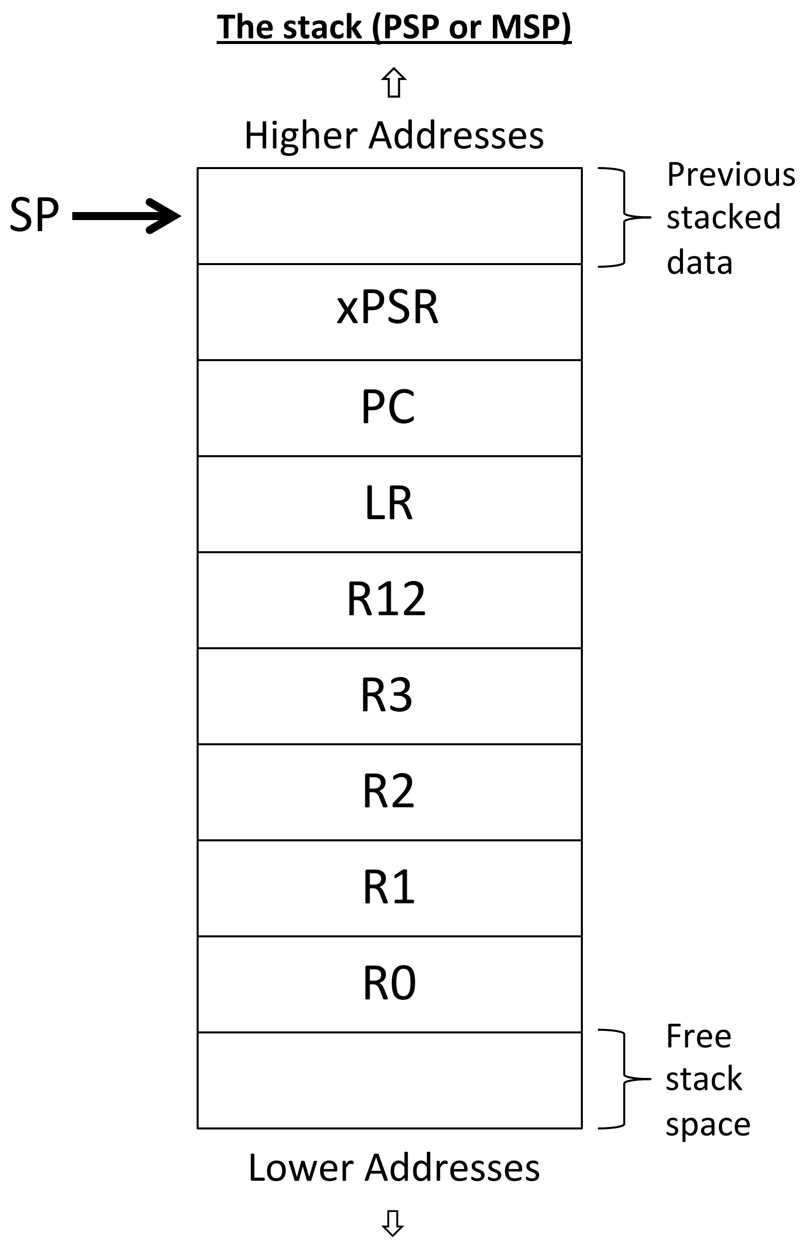
- Over 100 interrupt sources
  - Power on reset, bus errors, I/O pins changing state, data in on a serial bus etc.
- Need a great deal of control
  - Ability to enable and disable interrupt sources
  - Ability to control where to branch to for each interrupt
  - Ability to set interrupt priorities
    - Who wins in case of a tie
    - Can interrupt **A** interrupt the ISR for interrupt **B**?
      - If so, **A** can "preempt" **B**.
- All that control will involve memory mapped I/O.
  - And given the number of interrupts that's going to be a pain

Basic interrupt processing

- Stacking
  - Automatically by CPU
  - Maintains ABI semantics
  - ISRs can be C functions

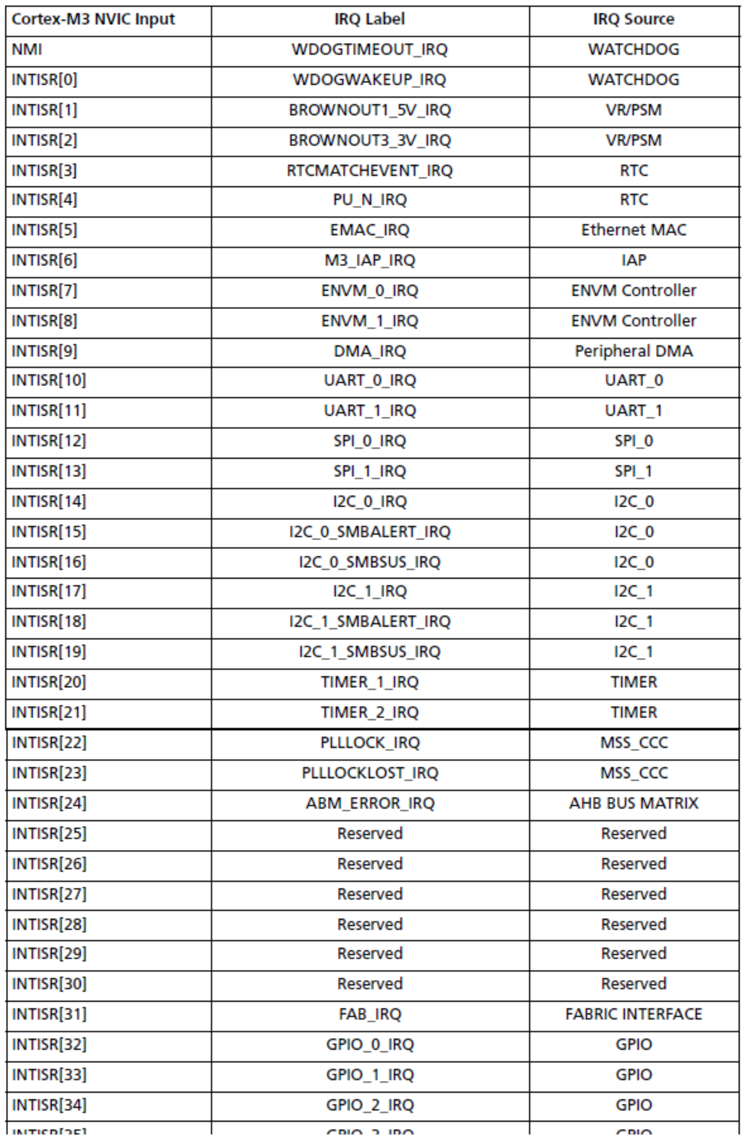
- Vector Fetch
  - We'll see this next

- Exit: update of SP, LR, PC

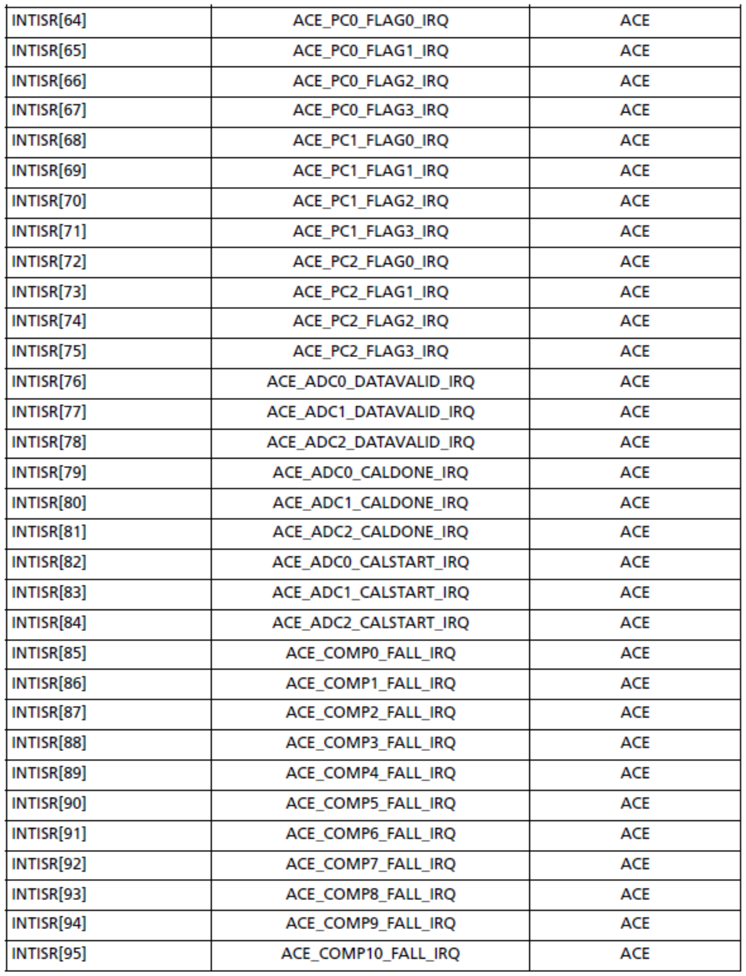


SmartFusion interrupt sources

Table 1-5 • SmartFusion Interrupt Sources



Cortex-M3 NVIC Input	IRQ Label	IRQ Source
NMI	WDOGTIMEOUT_IRQ	WATCHDOG
INTISR[0]	WDOGWAKEUP_IRQ	WATCHDOG
INTISR[1]	BROWNOUT1_5V_IRQ	VR/PSM
INTISR[2]	BROWNOUT3_3V_IRQ	VR/PSM
INTISR[3]	RTCMATCHEVENT_IRQ	RTC
INTISR[4]	PU_N_IRQ	RTC
INTISR[5]	EMAC_IRQ	Ethernet MAC
INTISR[6]	M3_IAP_IRQ	IAP
INTISR[7]	ENVM_0_IRQ	ENVM Controller
INTISR[8]	ENVM_1_IRQ	ENVM Controller
INTISR[9]	DMA_IRQ	Peripheral DMA
INTISR[10]	UART_0_IRQ	UART_0
INTISR[11]	UART_1_IRQ	UART_1
INTISR[12]	SPI_0_IRQ	SPI_0
INTISR[13]	SPI_1_IRQ	SPI_1
INTISR[14]	I2C_0_IRQ	I2C_0
INTISR[15]	I2C_0_SMBALERT_IRQ	I2C_0
INTISR[16]	I2C_0_SMBSUS_IRQ	I2C_0
INTISR[17]	I2C_1_IRQ	I2C_1
INTISR[18]	I2C_1_SMBALERT_IRQ	I2C_1
INTISR[19]	I2C_1_SMBSUS_IRQ	I2C_1
INTISR[20]	TIMER_1_IRQ	TIMER
INTISR[21]	TIMER_2_IRQ	TIMER
INTISR[22]	PLLLOCK_IRQ	MSS_CCC
INTISR[23]	PLLLOCKLOST_IRQ	MSS_CCC
INTISR[24]	ABM_ERROR_IRQ	AHB BUS MATRIX
INTISR[25]	Reserved	Reserved
INTISR[26]	Reserved	Reserved
INTISR[27]	Reserved	Reserved
INTISR[28]	Reserved	Reserved
INTISR[29]	Reserved	Reserved
INTISR[30]	Reserved	Reserved
INTISR[31]	FAB_IRQ	FABRIC INTERFACE
INTISR[32]	GPIO_0_IRQ	GPIO
INTISR[33]	GPIO_1_IRQ	GPIO
INTISR[34]	GPIO_2_IRQ	GPIO



INTISR[64]	ACE_PC0_FLAG0_IRQ	ACE
INTISR[65]	ACE_PC0_FLAG1_IRQ	ACE
INTISR[66]	ACE_PC0_FLAG2_IRQ	ACE
INTISR[67]	ACE_PC0_FLAG3_IRQ	ACE
INTISR[68]	ACE_PC1_FLAG0_IRQ	ACE
INTISR[69]	ACE_PC1_FLAG1_IRQ	ACE
INTISR[70]	ACE_PC1_FLAG2_IRQ	ACE
INTISR[71]	ACE_PC1_FLAG3_IRQ	ACE
INTISR[72]	ACE_PC2_FLAG0_IRQ	ACE
INTISR[73]	ACE_PC2_FLAG1_IRQ	ACE
INTISR[74]	ACE_PC2_FLAG2_IRQ	ACE
INTISR[75]	ACE_PC2_FLAG3_IRQ	ACE
INTISR[76]	ACE_ADC0_DATAVALID_IRQ	ACE
INTISR[77]	ACE_ADC1_DATAVALID_IRQ	ACE
INTISR[78]	ACE_ADC2_DATAVALID_IRQ	ACE
INTISR[79]	ACE_ADC0_CALDONE_IRQ	ACE
INTISR[80]	ACE_ADC1_CALDONE_IRQ	ACE
INTISR[81]	ACE_ADC2_CALDONE_IRQ	ACE
INTISR[82]	ACE_ADC0_CALSTART_IRQ	ACE
INTISR[83]	ACE_ADC1_CALSTART_IRQ	ACE
INTISR[84]	ACE_ADC2_CALSTART_IRQ	ACE
INTISR[85]	ACE_COMP0_FALL_IRQ	ACE
INTISR[86]	ACE_COMP1_FALL_IRQ	ACE
INTISR[87]	ACE_COMP2_FALL_IRQ	ACE
INTISR[88]	ACE_COMP3_FALL_IRQ	ACE
INTISR[89]	ACE_COMP4_FALL_IRQ	ACE
INTISR[90]	ACE_COMP5_FALL_IRQ	ACE
INTISR[91]	ACE_COMP6_FALL_IRQ	ACE
INTISR[92]	ACE_COMP7_FALL_IRQ	ACE
INTISR[93]	ACE_COMP8_FALL_IRQ	ACE
INTISR[94]	ACE_COMP9_FALL_IRQ	ACE
INTISR[95]	ACE_COMP10_FALL_IRQ	ACE

54 more ACE specific interrupts

GPIO_3_IRQ to GPIO_31_IRQ cut

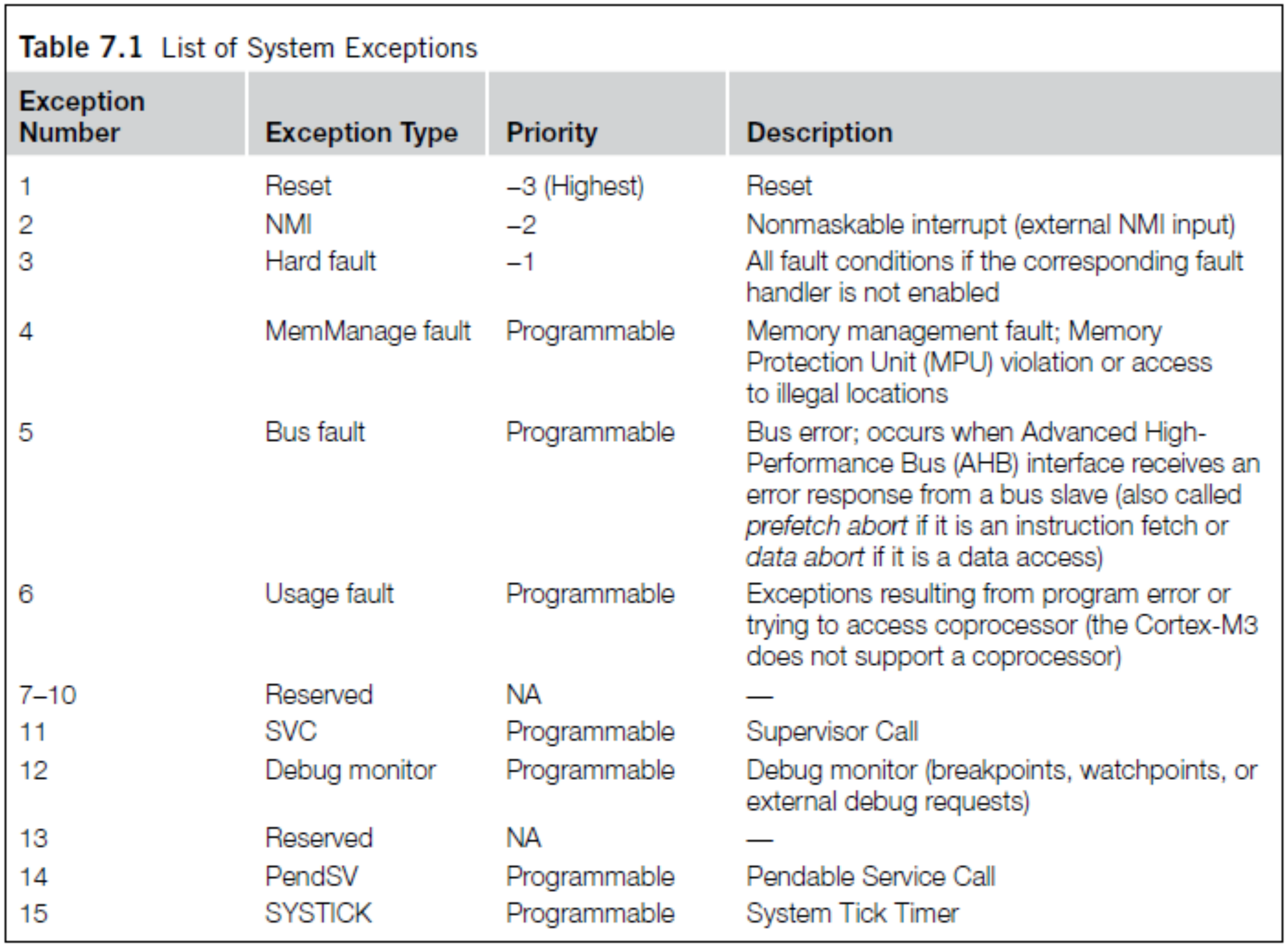


Table 7.1 List of System Exceptions

Exception Number	Exception Type	Priority	Description
1	Reset	–3 (Highest)	Reset
2	NMI	–2	Nonmaskable interrupt (external NMI input)
3	Hard fault	–1	All fault conditions if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; Memory Protection Unit (MPU) violation or access to illegal locations
5	Bus fault	Programmable	Bus error; occurs when Advanced High-Performance Bus (AHB) interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch or *data abort* if it is a data access)
6	Usage fault	Programmable	Exceptions resulting from program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	—
11	SVC	Programmable	Supervisor Call
12	Debug monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	—
14	PendSV	Programmable	Pendable Service Call
15	SYSTICK	Programmable	System Tick Timer

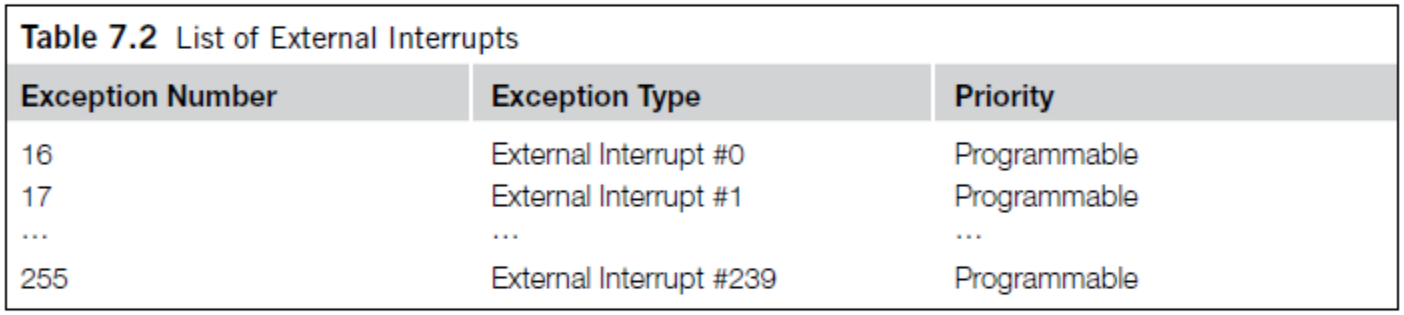


Table 7.2 List of External Interrupts

Exception Number	Exception Type	Priority
16	External Interrupt #0	Programmable
17	External Interrupt #1	Programmable
...	...	...
255	External Interrupt #239	Programmable

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)

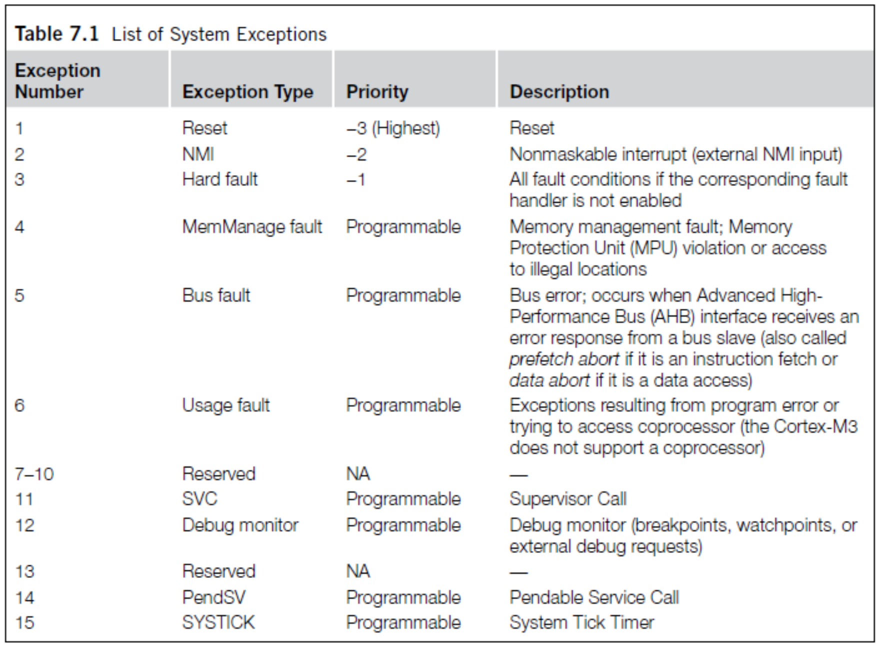


Table 7.1 List of System Exceptions

Exception Number	Exception Type	Priority	Description
1	Reset	–3 (Highest)	Reset
2	NMI	–2	Nonmaskable interrupt (external NMI input)
3	Hard fault	–1	All fault conditions if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; Memory Protection Unit (MPU) violation or access to illegal locations
5	Bus fault	Programmable	Bus error; occurs when Advanced High-Performance Bus (AHB) interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch or *data abort* if it is a data access)
6	Usage fault	Programmable	Exceptions resulting from program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	—
11	SVC	Programmable	Supervisor Call
12	Debug monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	—
14	PendSV	Programmable	Pendable Service Call
15	SYSTICK	Programmable	System Tick Timer

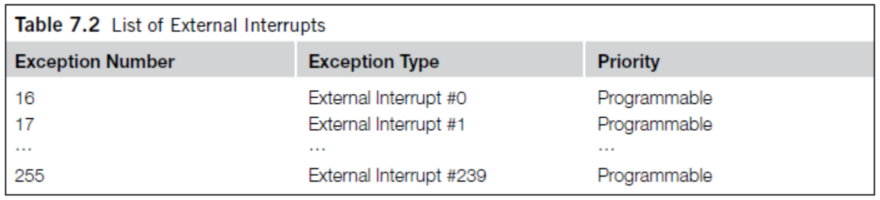
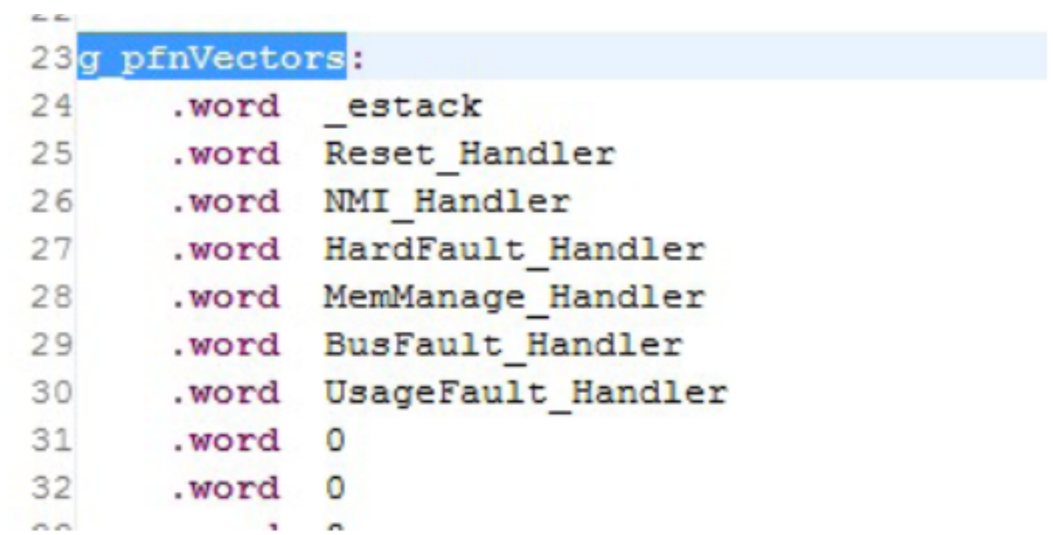


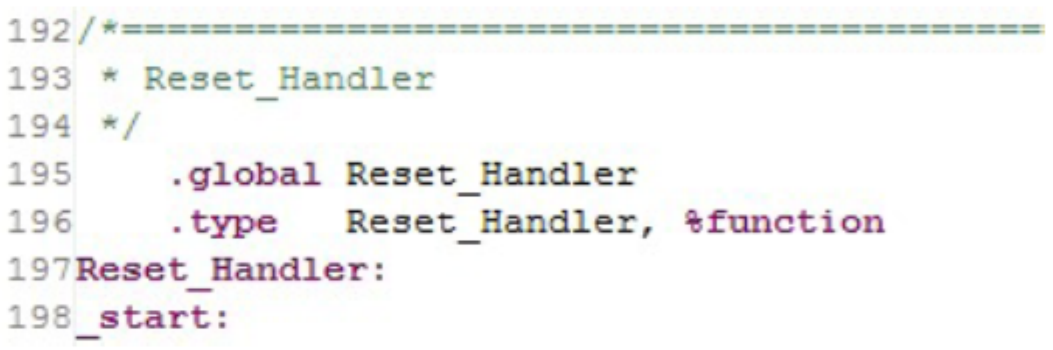
Table 7.2 List of External Interrupts

Exception Number	Exception Type	Priority
16	External Interrupt #0	Programmable
17	External Interrupt #1	Programmable
…	…	…
255	External Interrupt #239	Programmable

How to change where to go on an interrupt?
Answer: edit the interrupt vector table [IVT]



```
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
```



```
/*==========================================
 * Reset_Handler
 */
    .global Reset_Handler
    .type   Reset_Handler, %function
Reset_Handler:
_start:
```

NVIC/Interrupt configuration registers

ICTR	Interrupt Controller Type Register (RW)
ISER	Interrupt Set-Enable Register (RW)
ICER	Interrupt Clear-Enable Register (RW)
ISPR	Interrupt Set-Pending Register (RW)
ICPR	Interrupt Clear-Pending Register (RW)
IABR	Interrupt Active Bit Register (RO)
IPR	Interrupt Priority Register (RW)
AIRC	Application Interrupt and Reset Control

Enabling and disabling interrupt sources

- Interrupt Set Enable and Clear Enable
  - 0xE000E100-0xE000E11C, 0xE000E180-0xE000E19C

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

0xE000E200	SETPEND0	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
0xE000E280	CLRPEND0	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

Address	Name	Type	Reset Value	Description
0xE000E300	ACTIVE0	R	0	Active status for external interrupt #0–31 bit[0] for interrupt #0 bit[1] for interrupt #1 ... bit[31] for interrupt #31
0xE000E304	ACTIVE1	R	0	Active status for external interrupt #32–63
...	–	–	–	–

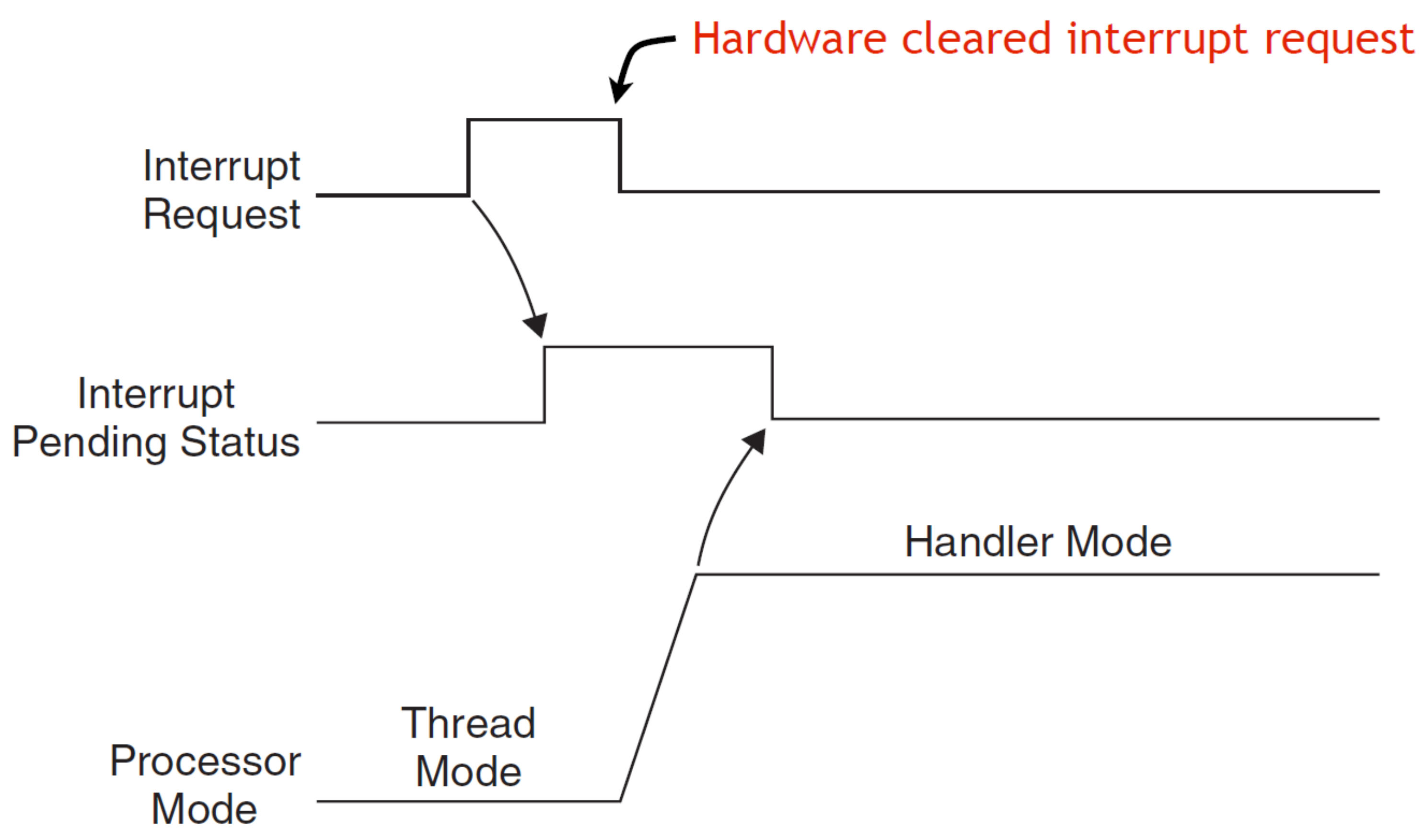
Exercise: Enabling interrupt sources

- Implement the following function to enable external interrupt #x when called:

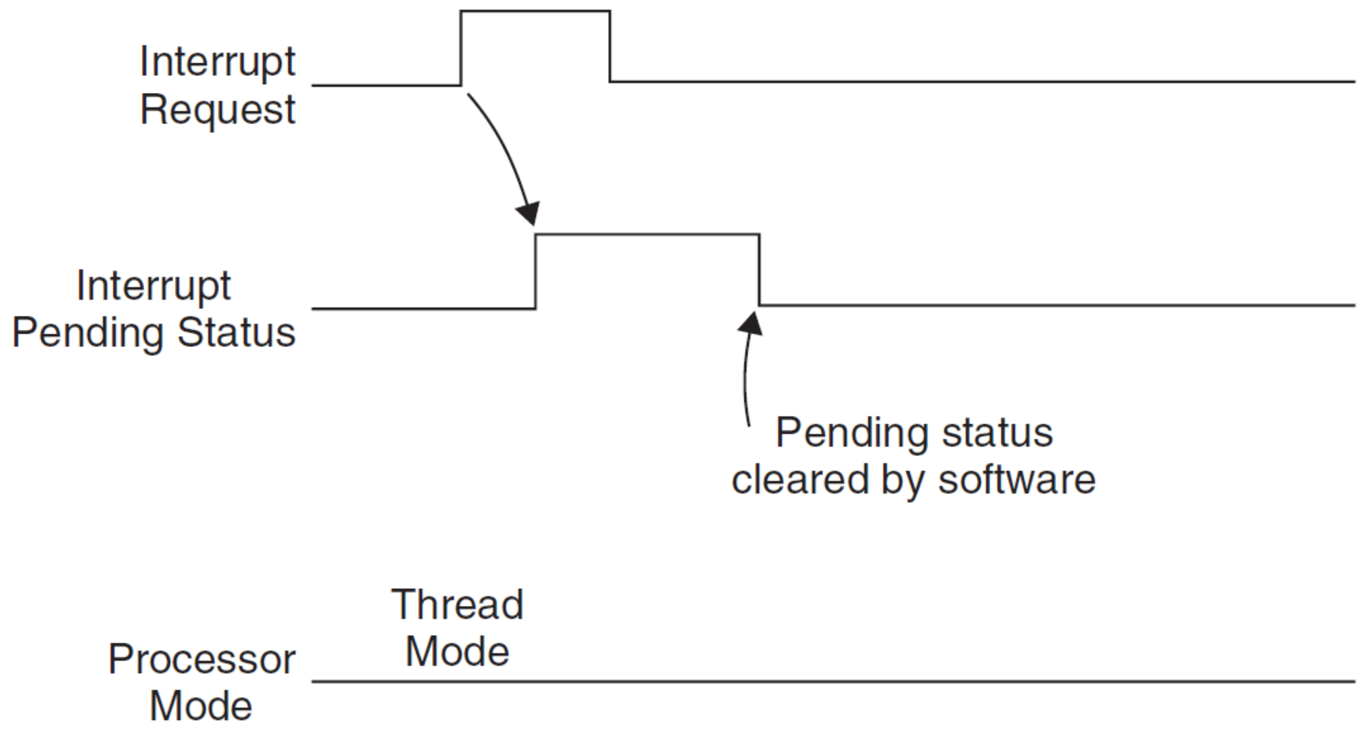
```
void enable_interrupt(int x) {
  /* your code here */
}
```

- Umm, so what do we have to do?
  - Use top (32-5)=27 bits of x to select the word offset
  - Offset from what, you ask? Base of ISER (0xE000E100)
  - Use the bottom five bits of x to select bit position
  - Write a ‘1’ to that bit position at memory addr=base+offset
  - You’re done!

Pending interrupts

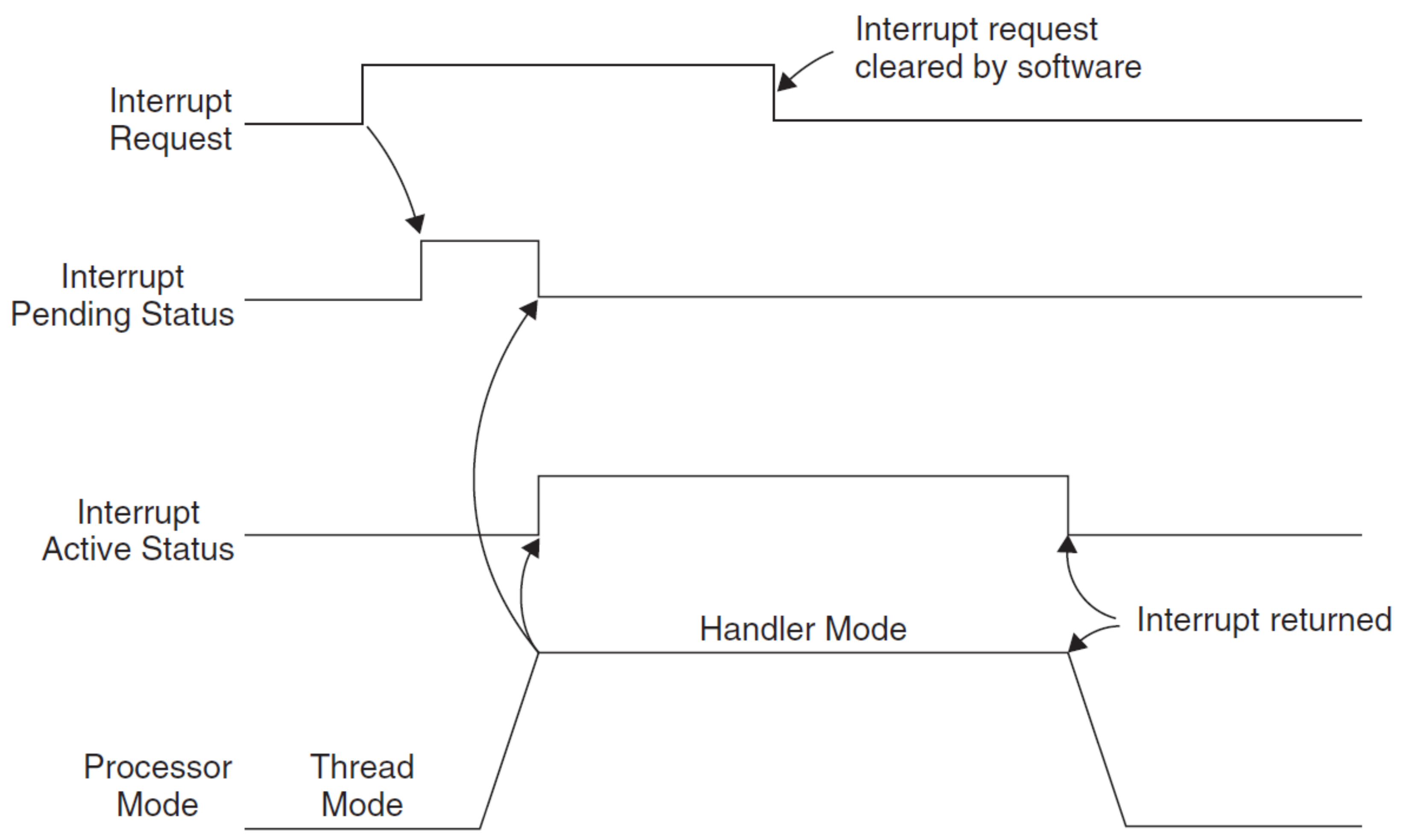


The normal case. Once Interrupt request is seen, processor puts it in "pending" state even if hardware drops the request.
IPS is cleared by the hardware once we jump to the ISR.

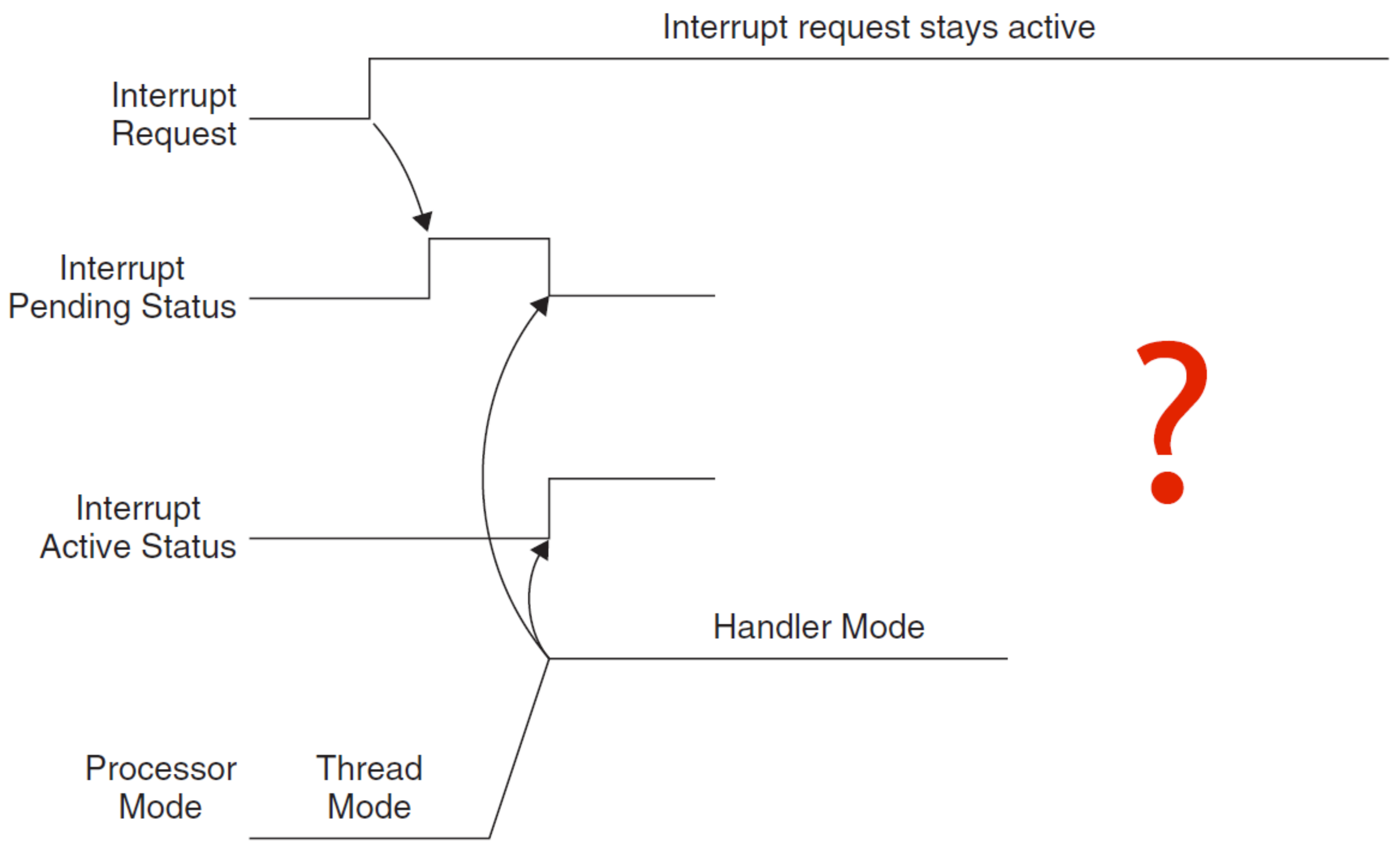


In this case, the processor never took the interrupt because we cleared the IPS by hand (via a memory-mapped I/O register)

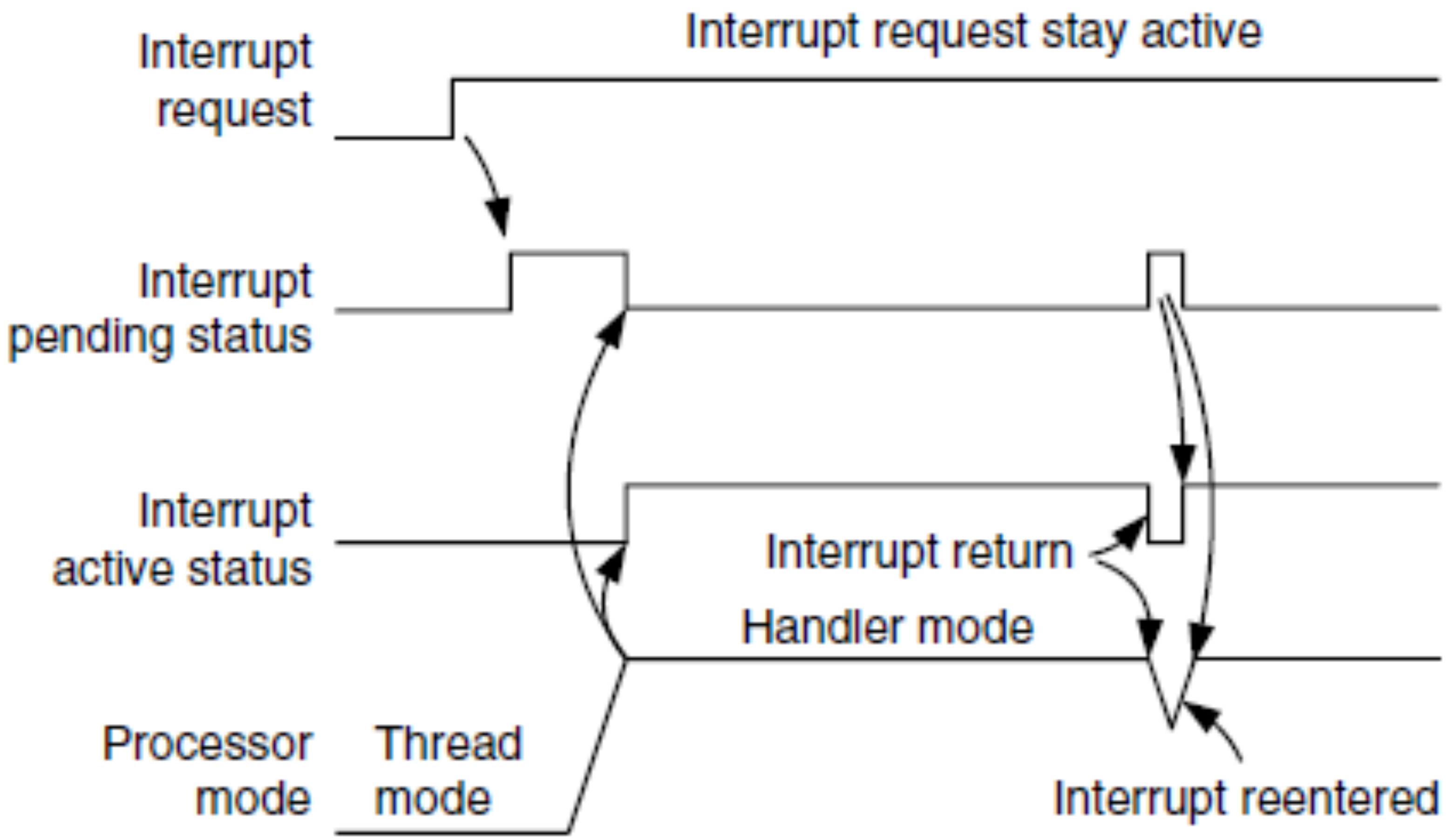
Active Status set during handler execution



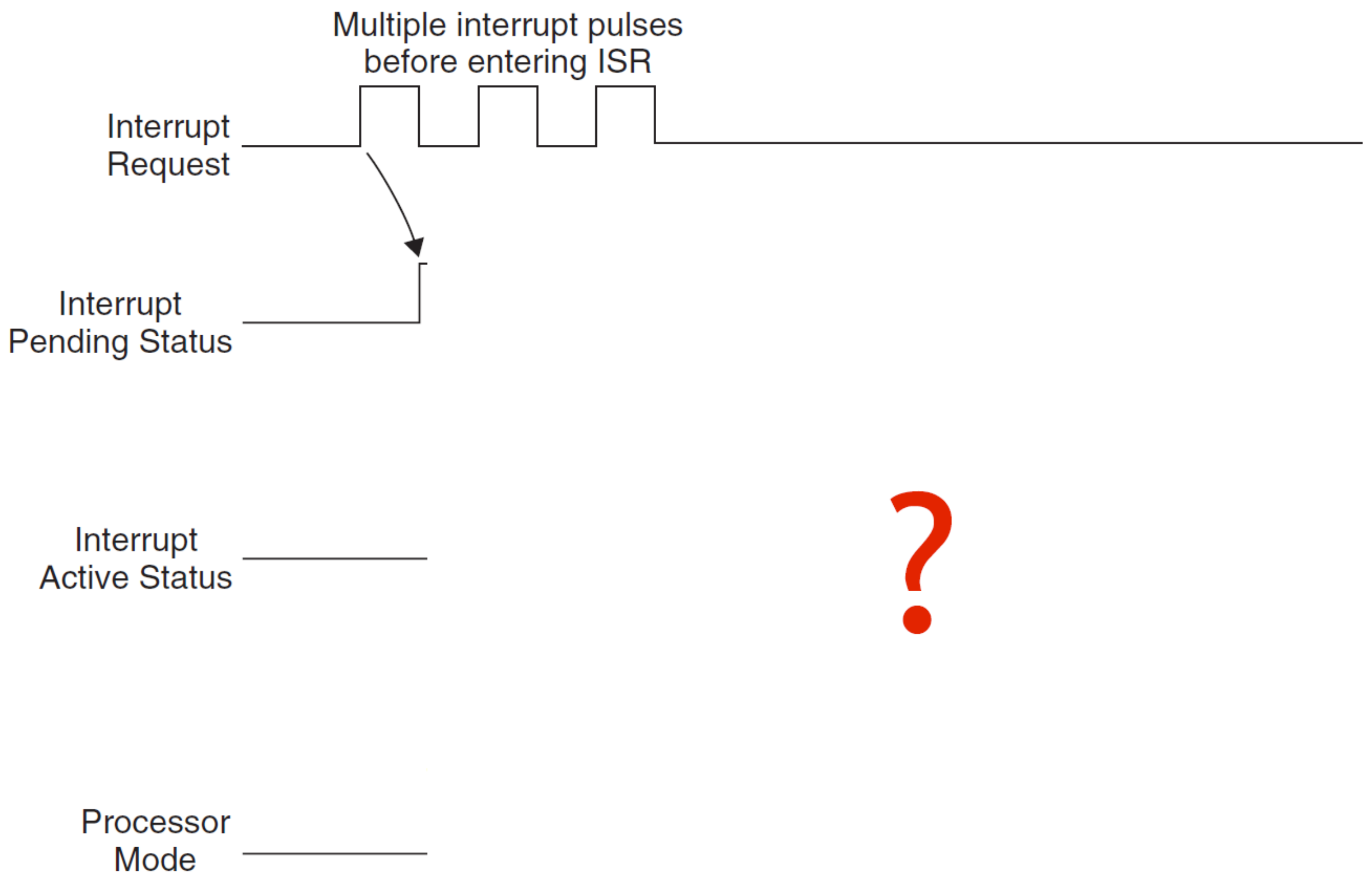
Interrupt Request not Cleared



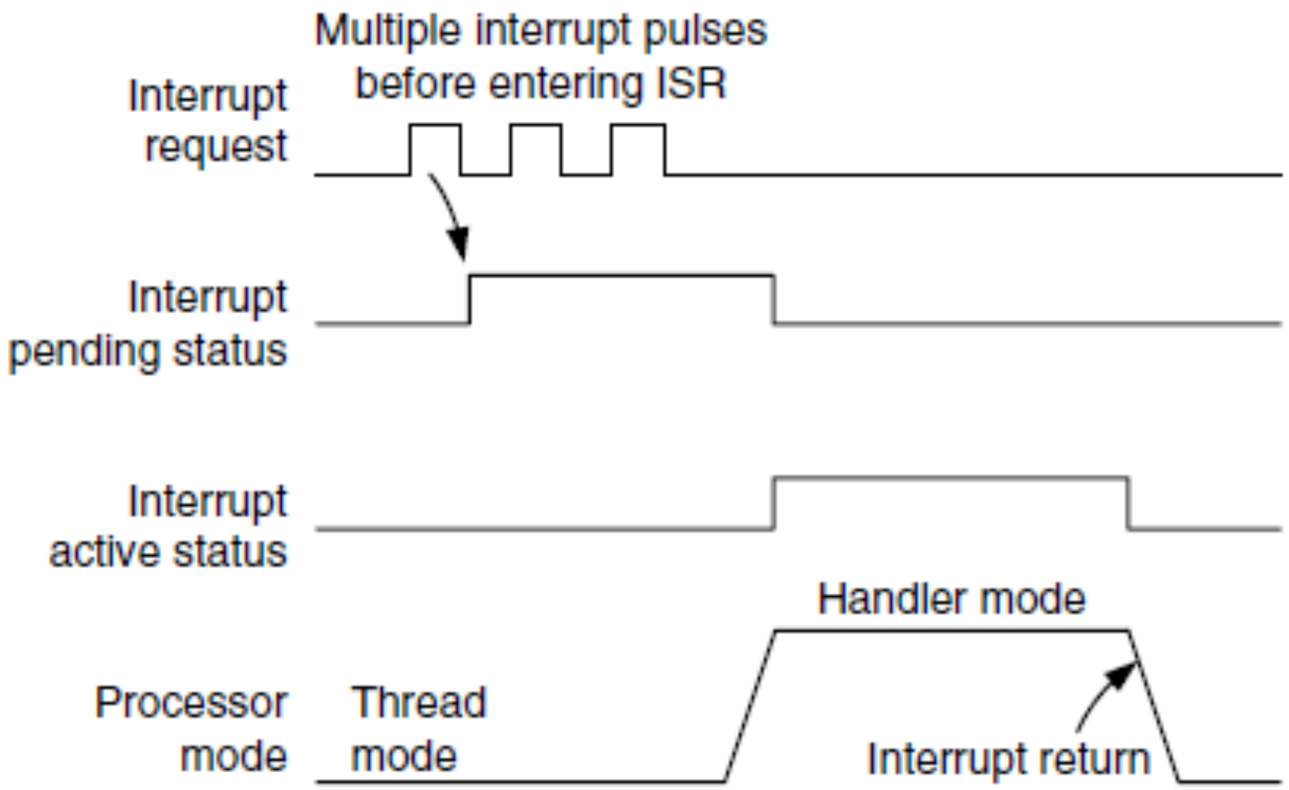
Answer


Interrupt request stay active
Interrupt request
Interrupt pending status
Interrupt active status
Interrupt return
Handler mode
Processor mode
Thread mode
Interrupt reentered

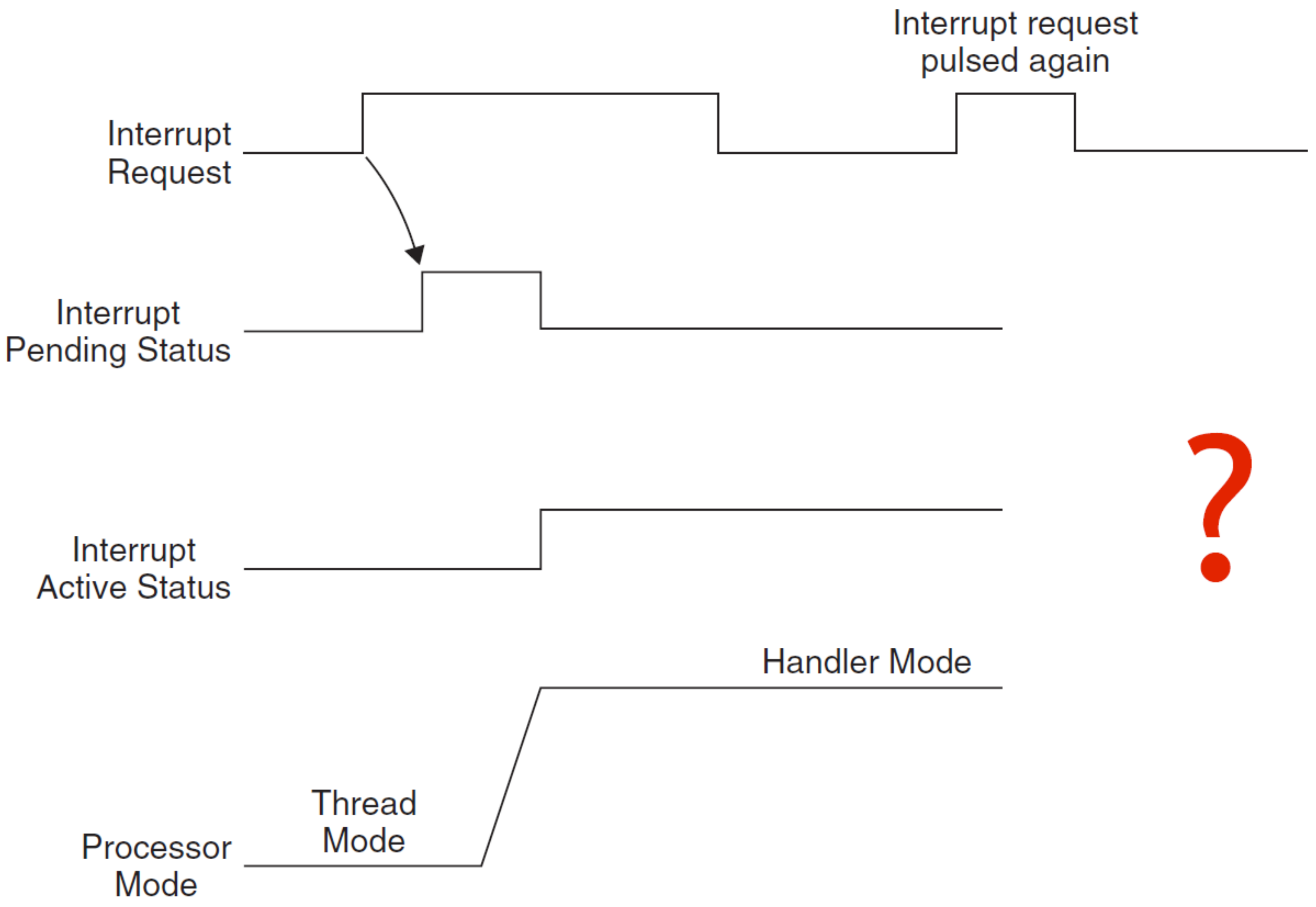
Interrupt pulses before entering ISR



Answer


Multiple interrupt pulses before entering ISR
Interrupt request
Interrupt pending status
Interrupt active status
Handler mode
Processor mode
Thread mode
Interrupt return

New Interrupt Request after Pending Cleared



Interrupt Priority

- What do we do if several interrupts arrive simultaneously?
- NVIC allows 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

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented			Not implemented, read as zero				

  - 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 & subpriority
  - Preempt priority: indicates if an interrupt can preempt another
  - Subpriority: used to determine which is served first if two interrupts of same group arrive concurrently

Interrupt Priority (2)

- Interrupt priority level registers
  - Range: 0xE000E400 to 0xE000E4EF

Address	Name	Type	Reset Value	Description
0xE000E400	PRI_0	R/W	0 (8-bit)	Priority-level external interrupt #0
0xE000E401	PRI_1	R/W	0 (8-bit)	Priority-level external interrupt #1
…	–	–	–	–
0xE000E41F	PRI_31	R/W	0 (8-bit)	Priority-level external interrupt #31
…	–	–	–	–

Preemption Priority and Subpriority

Priority Group	Preempt Priority Field	Subpriority Field
0	Bit [7:1]	Bit [0]
1	Bit [7:2]	Bit [1:0]
2	Bit [7:3]	Bit [2:0]
3	Bit [7:4]	Bit [3:0]
4	Bit [7:5]	Bit [4:0]
5	Bit [7:6]	Bit [5:0]
6	Bit [7]	Bit [6:0]
7	None	Bit [7:0]

Application Interrupt and Reset Control Register (Address 0xE000ED0C)

Bits	Name	Type	Reset Value	Description
31:16	VECTKEY	R/W	–	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
15	ENDIANNESS	R	–	Indicates endianness for data: 1 for big endian (BE8) and 0 for little endian; this can only change after a reset
10:8	PRIGROUP	R/W	0	Priority group
2	SYSRESETREQ	W	–	Requests chip control logic to generate a reset
1	VECTCLRACTIVE	W	–	Clears all active state information for exceptions; typically used in debug or OS to allow system to recover from system error (Reset is safer)
0	VECTRESET	W	–	Resets the Cortex-M3 processor (except debug logic), but this will not reset circuits outside the processor

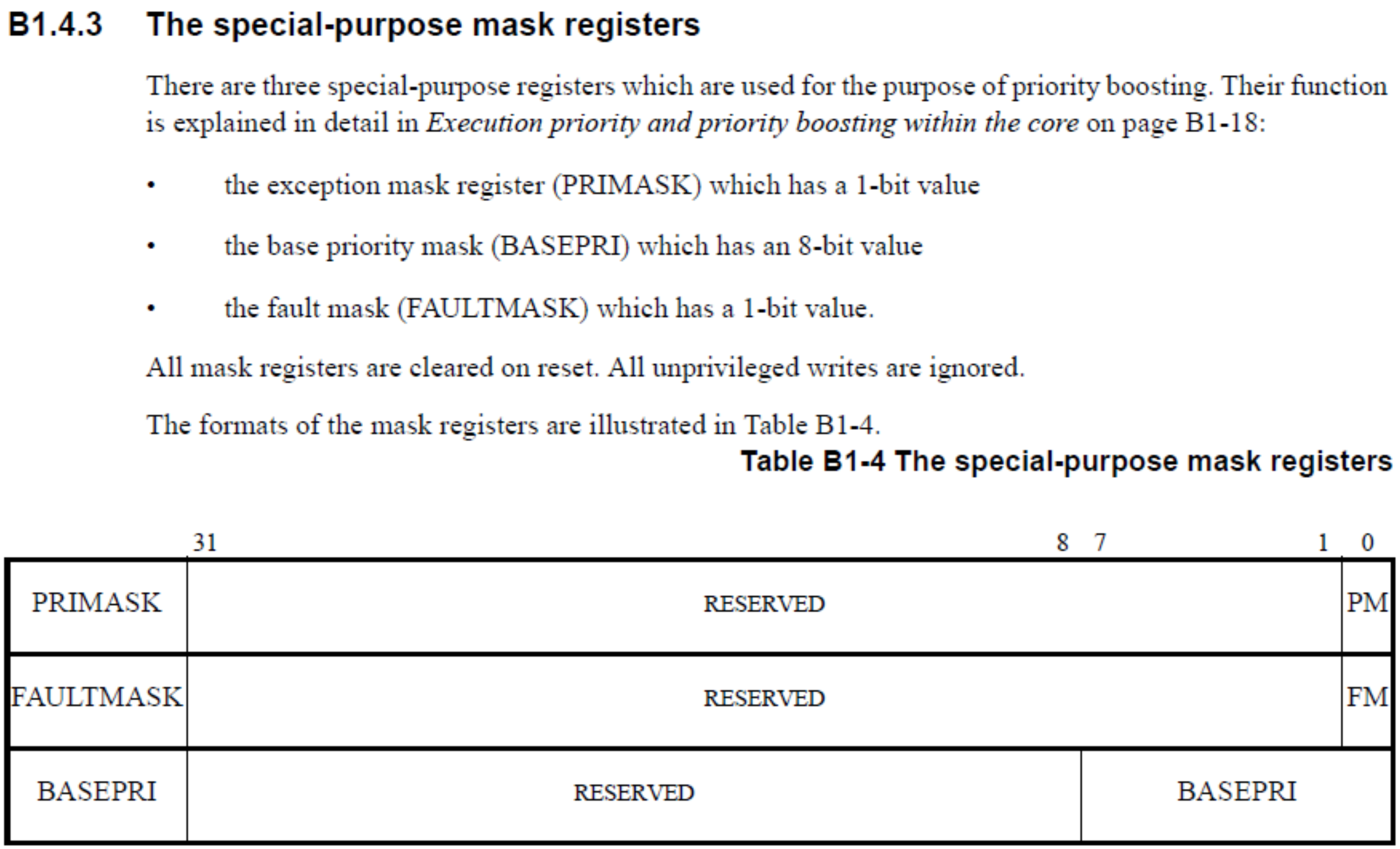
PRIMASK, FAULTMASK, and BASEPRI registers

- What if we quickly want to disable all interrupts?
- Write 1 into PRIMASK to disable all interrupts except NMI
  - MOV R0, #1
  - MSR PRIMASK, R0 ; MSR and MRS are special instructions
- Write 0 into PRIMASK to enable all interrupts

- FAULTMASK is the same as PRIMASK, but it also blocks hard faults (priority = -1)

- What if we want to disable all interrupts below a certain priority?
- Write priority into BASEPRI register
  - MOV R0, #0x60
  - MSR BASEPRI, R0

Masking



B1.4.3 The special-purpose mask registers

There are three special-purpose registers which are used for the purpose of priority boosting. Their function is explained in detail in *Execution priority and priority boosting within the core* on page B1-18:

- the exception mask register (PRIMASK) which has a 1-bit value
- the base priority mask (BASEPRI) which has an 8-bit value
- the fault mask (FAULTMASK) which has a 1-bit value.

All mask registers are cleared on reset. All unprivileged writes are ignored.

The formats of the mask registers are illustrated in Table B1-4.

Table B1-4 The special-purpose mask registers

	31 8	7 1	0
PRIMASK	RESERVED	RESERVED	PM
FAULTMASK	RESERVED	RESERVED	FM
BASEPRI	RESERVED	BASEPRI	BASEPRI

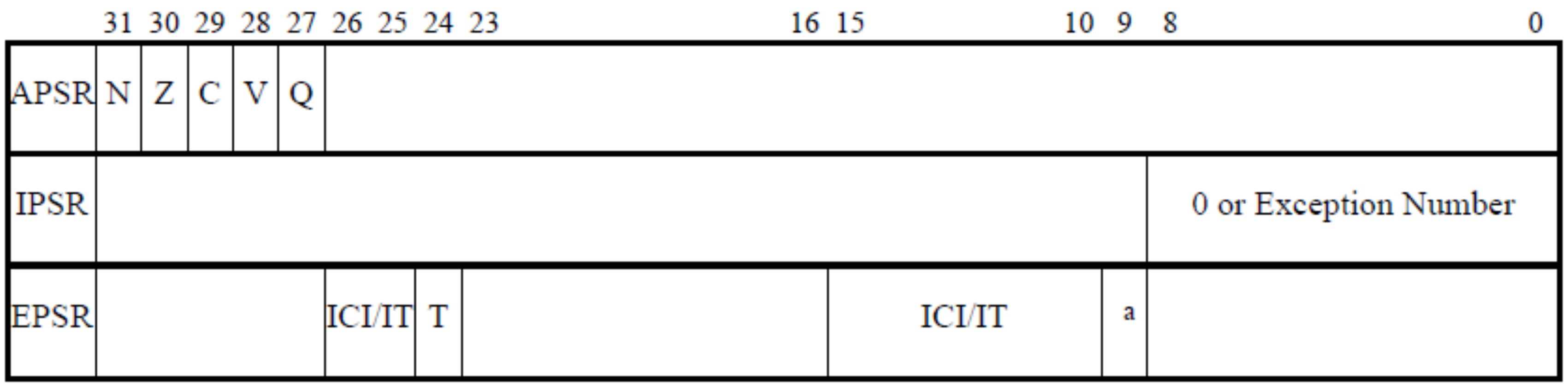
Interrupt Service Routines

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

Table B1-2 The xPSR register layout



	31	30	29	28	27	26 25	24	23 16	15 10	9	8 0
APSR	N	Z	C	V	Q						
IPSR											0 or Exception Number
EPSR						ICI/IT	T		ICI/IT	a	

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)

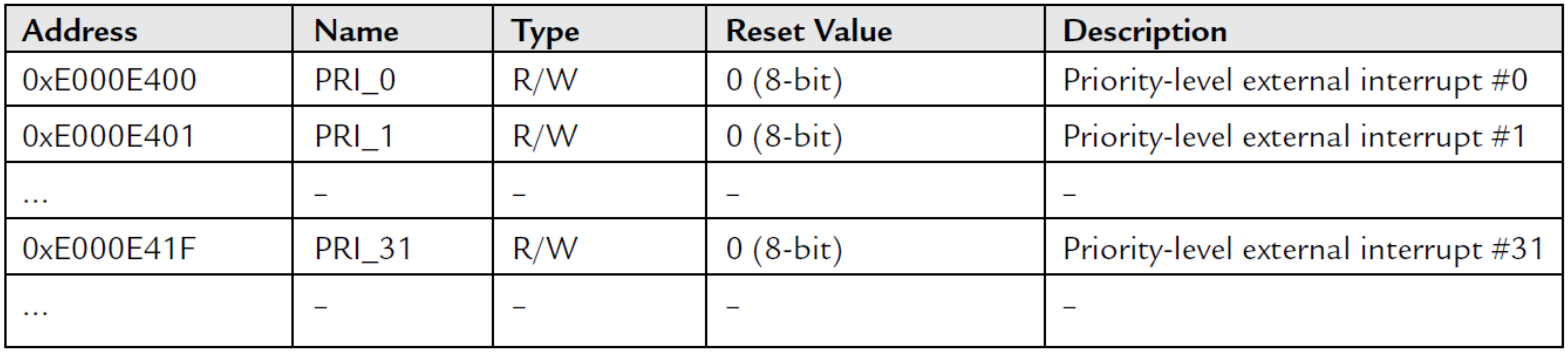
- Set Pending & Clear Pending
  - 0xE000E200-0xE000E21C, 0xE000E280-0xE000E29C

0xE000E200	SETPEND0	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

0xE000E280	CLRPEND0	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

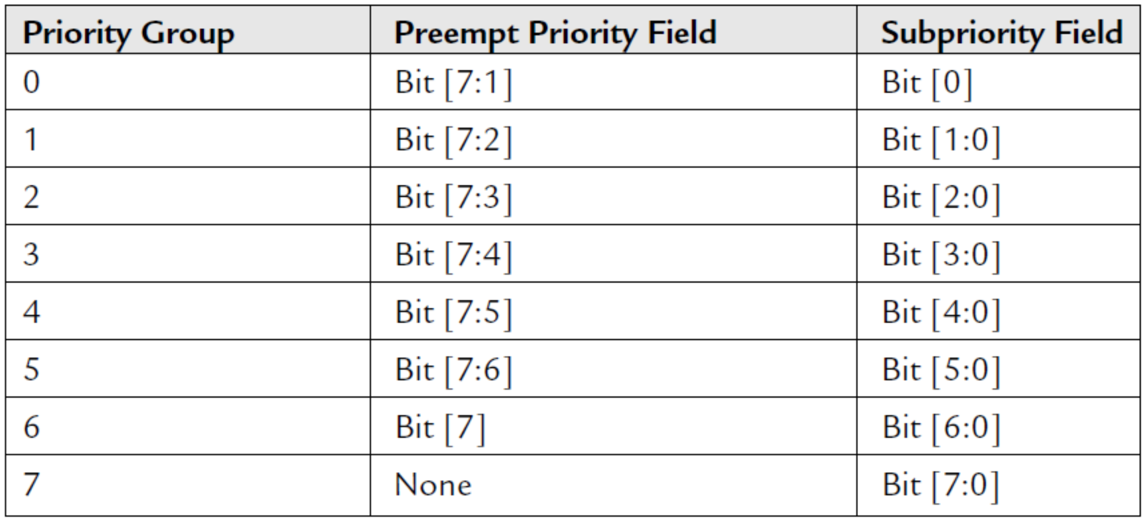
1. More registers (example)

- Interrupt Priority Level Registers
  - 0xE000E400-0xE000E4EF



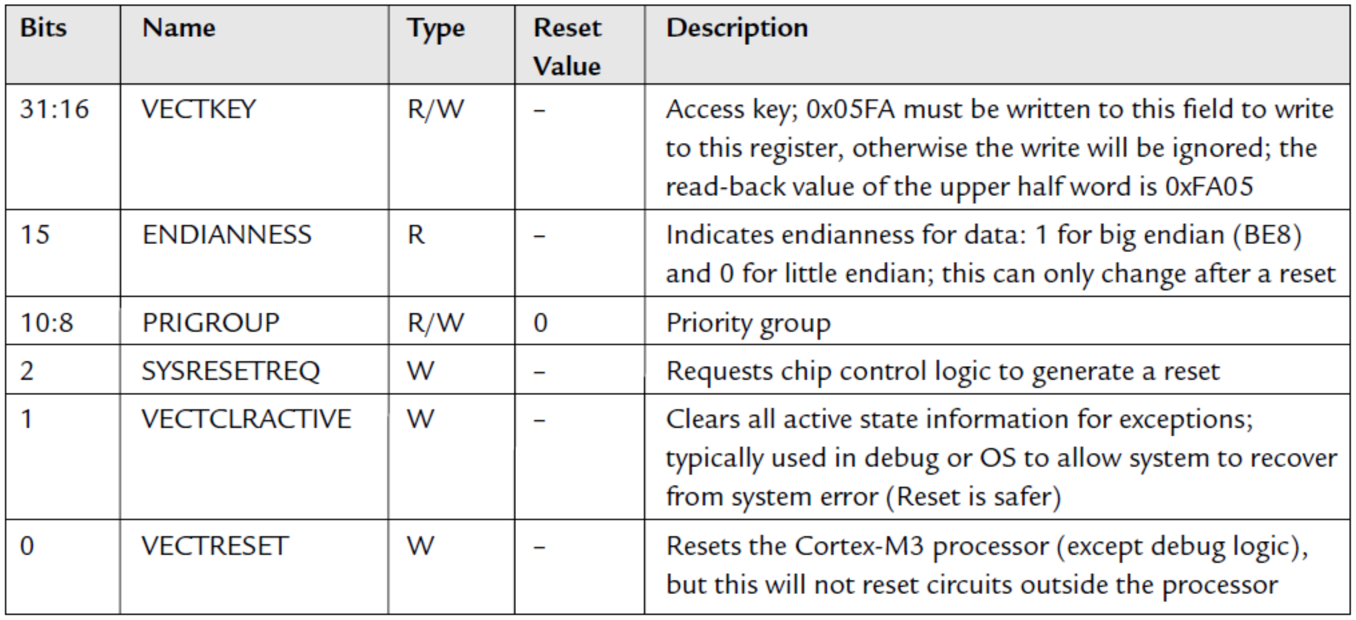
Address	Name	Type	Reset Value	Description
0xE000E400	PRI_0	R/W	0 (8-bit)	Priority-level external interrupt #0
0xE000E401	PRI_1	R/W	0 (8-bit)	Priority-level external interrupt #1
…	–	–	–	–
0xE000E41F	PRI_31	R/W	0 (8-bit)	Priority-level external interrupt #31
…	–	–	–	–

1. Yet another part of the NVIC registers!



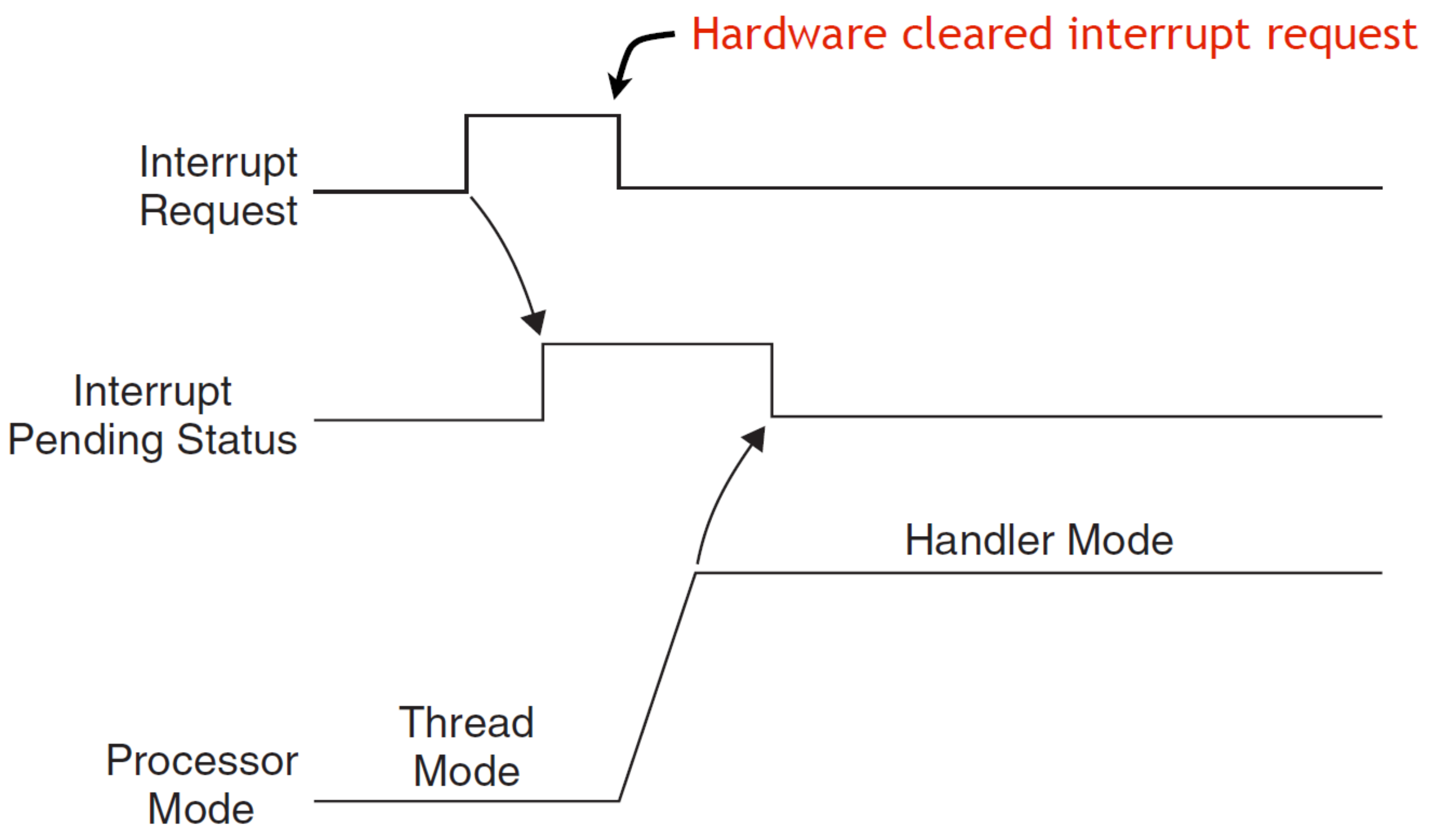
Priority Group	Preempt Priority Field	Subpriority Field
0	Bit [7:1]	Bit [0]
1	Bit [7:2]	Bit [1:0]
2	Bit [7:3]	Bit [2:0]
3	Bit [7:4]	Bit [3:0]
4	Bit [7:5]	Bit [4:0]
5	Bit [7:6]	Bit [5:0]
6	Bit [7]	Bit [6:0]
7	None	Bit [7:0]

Application Interrupt and Reset Control Register (Address 0xE000ED0C)



Bits	Name	Type	Reset Value	Description
31:16	VECTKEY	R/W	–	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
15	ENDIANNESS	R	–	Indicates endianness for data: 1 for big endian (BE8) and 0 for little endian; this can only change after a reset
10:8	PRIGROUP	R/W	0	Priority group
2	SYSRESETREQ	W	–	Requests chip control logic to generate a reset
1	VECTCLRACTIVE	W	–	Clears all active state information for exceptions; typically used in debug or OS to allow system to recover from system error (Reset is safer)
0	VECTRESET	W	–	Resets the Cortex-M3 processor (except debug logic), but this will not reset circuits outside the processor

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

```
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)
```

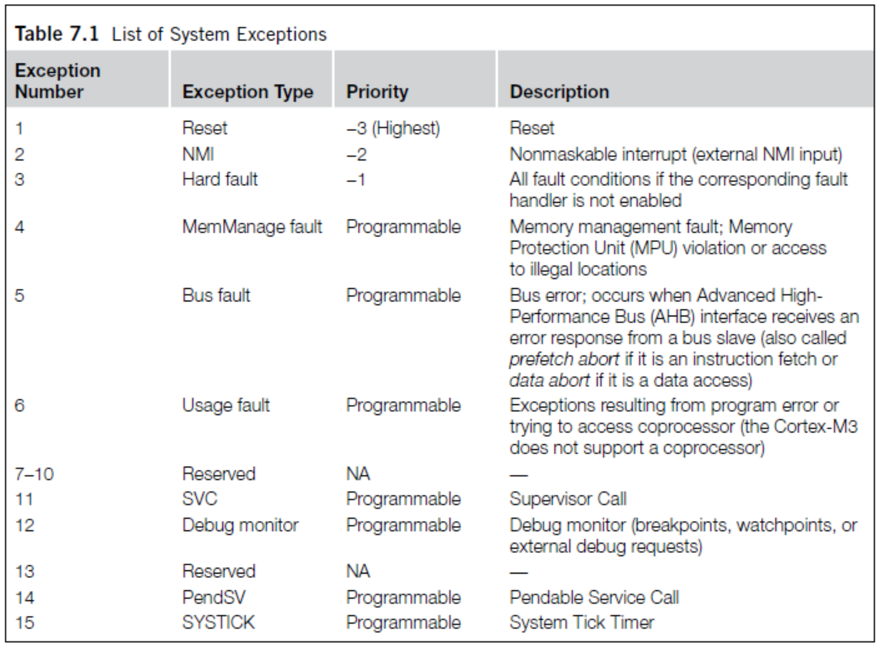


Table 7.1 List of System Exceptions

Exception Number	Exception Type	Priority	Description
1	Reset	–3 (Highest)	Reset
2	NMI	–2	Nonmaskable interrupt (external NMI input)
3	Hard fault	–1	All fault conditions if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; Memory Protection Unit (MPU) violation or access to illegal locations
5	Bus fault	Programmable	Bus error; occurs when Advanced High-Performance Bus (AHB) interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch or *data abort* if it is a data access)
6	Usage fault	Programmable	Exceptions resulting from program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	—
11	SVC	Programmable	Supervisor Call
12	Debug monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	—
14	PendSV	Programmable	Pendable Service Call
15	SYSTICK	Programmable	System Tick Timer

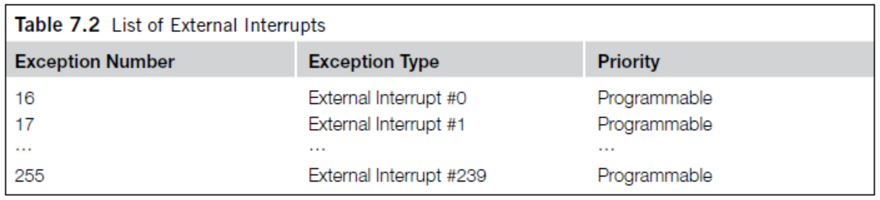


Table 7.2 List of External Interrupts

Exception Number	Exception Type	Priority
16	External Interrupt #0	Programmable
17	External Interrupt #1	Programmable
...	...	...
255	External Interrupt #239	Programmable

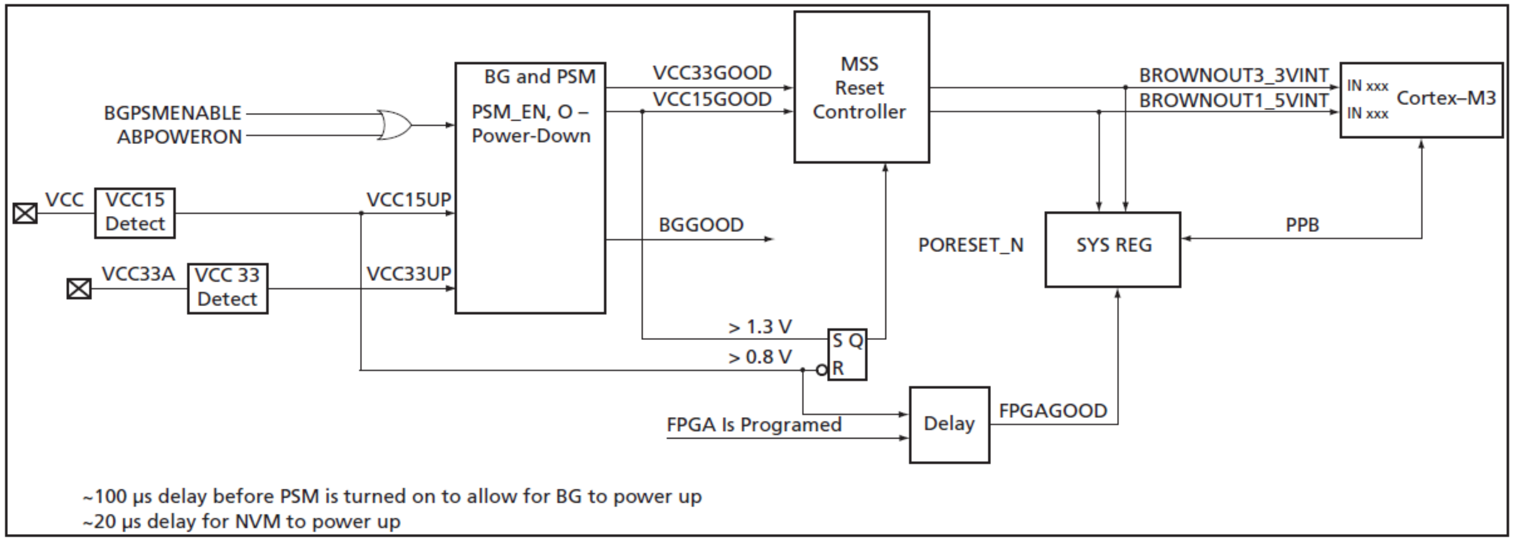
Discussion: So let’s say a GPIO pin goes high

- 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 (0xFFFFFFFx)
- 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 hander 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

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