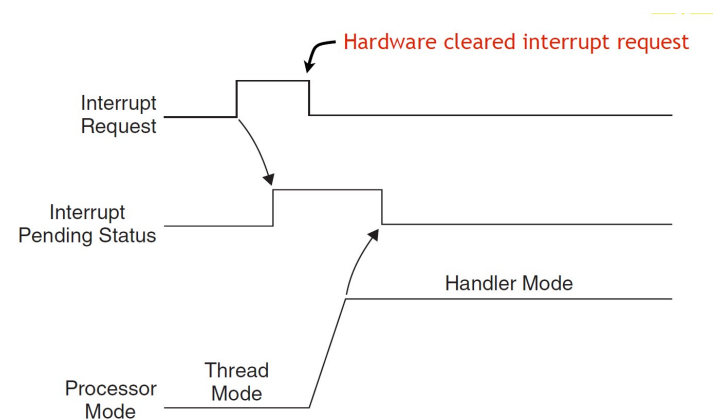


Prabal Dutta
University of Michigan

Modified by Jim Huang <jserv.tw@gmail.com>

Exceptions and 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 than it looks.
 - Why?

... 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?
- 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

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...

- 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?

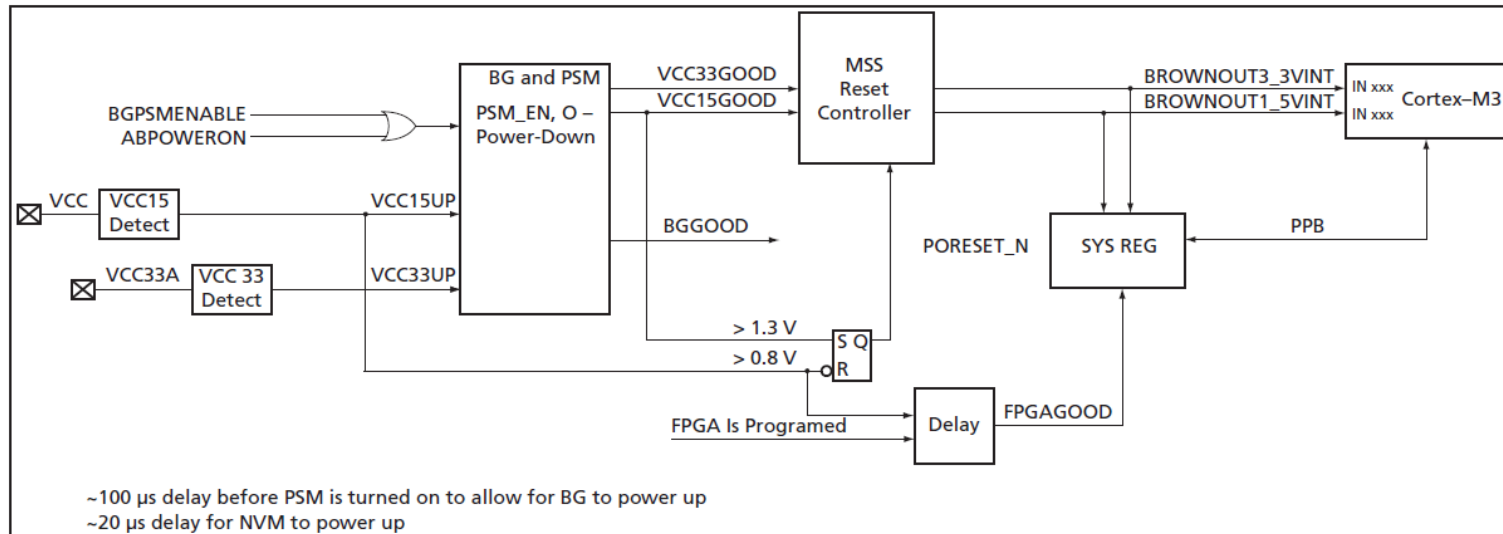
- If we get one interrupt while handling another what to do?
 - Just handle it
 - But what about that dedicated register?
 - What if I'm doing something that can't be stopped?
 - Ignore it
 - But what if it is important?
 - Prioritize
 - Take those interrupts you care about. Ignore the rest
 - Still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

Some interrupts...



Table 1-5 • SmartFusion Interrupt Sources

Cortex-M3 NVIC Input	IRQ Label	IRQ Source
NMI	WDGTIMEOUT_IRQ	WATCHDOG
INTISR[0]	WDGWAKEUP_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_SMBUS_IRQ	I2C_0
INTISR[17]	I2C_1_IRQ	I2C_1
INTISR[18]	I2C_1_SMBALERT_IRQ	I2C_1
INTISR[19]	I2C_1_SMBUS_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[35]	GPIO_3_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

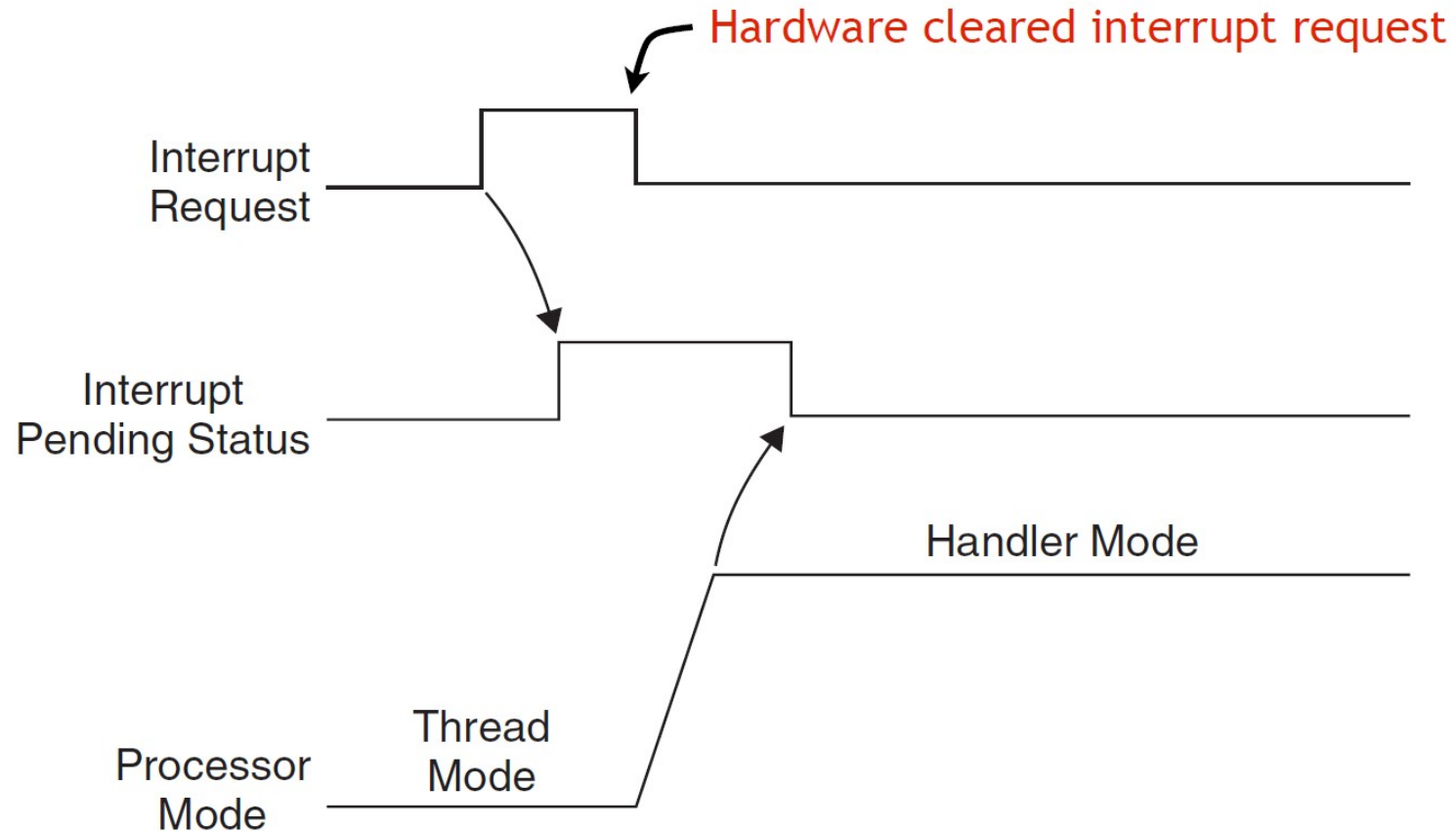
And the interrupt vectors

(in startup_a2fxxm3.s found in CMSIS, startup_gcc)



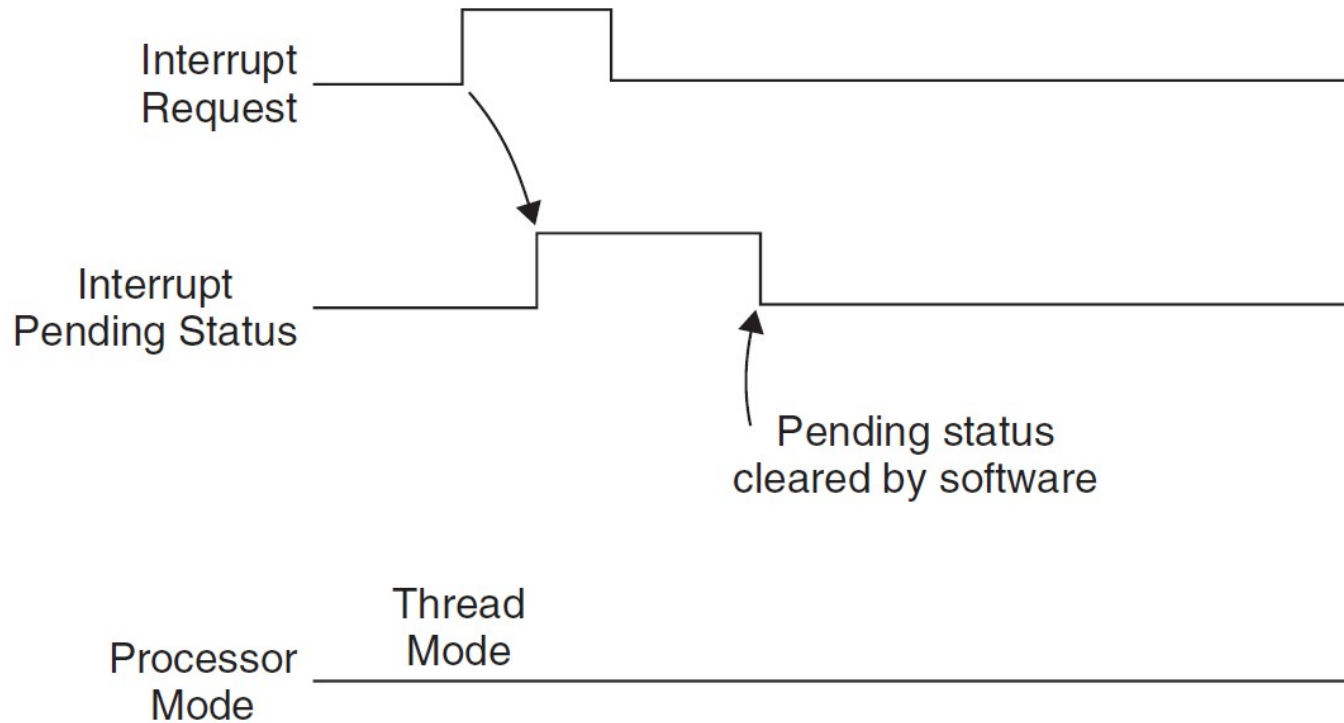
```
g_pfnVectors:
    .word  _estack
    .word  Reset_Handler
    .word  NMI_Handler
    .word  HardFault_Handler
    .word  MemManage_Handler
    .word  BusFault_Handler
    .word  UsageFault_Handler
    .word  0
    .word  0
    .word  0
    .word  0
    .word  SVC_Handler
    .word  DebugMon_Handler
    .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



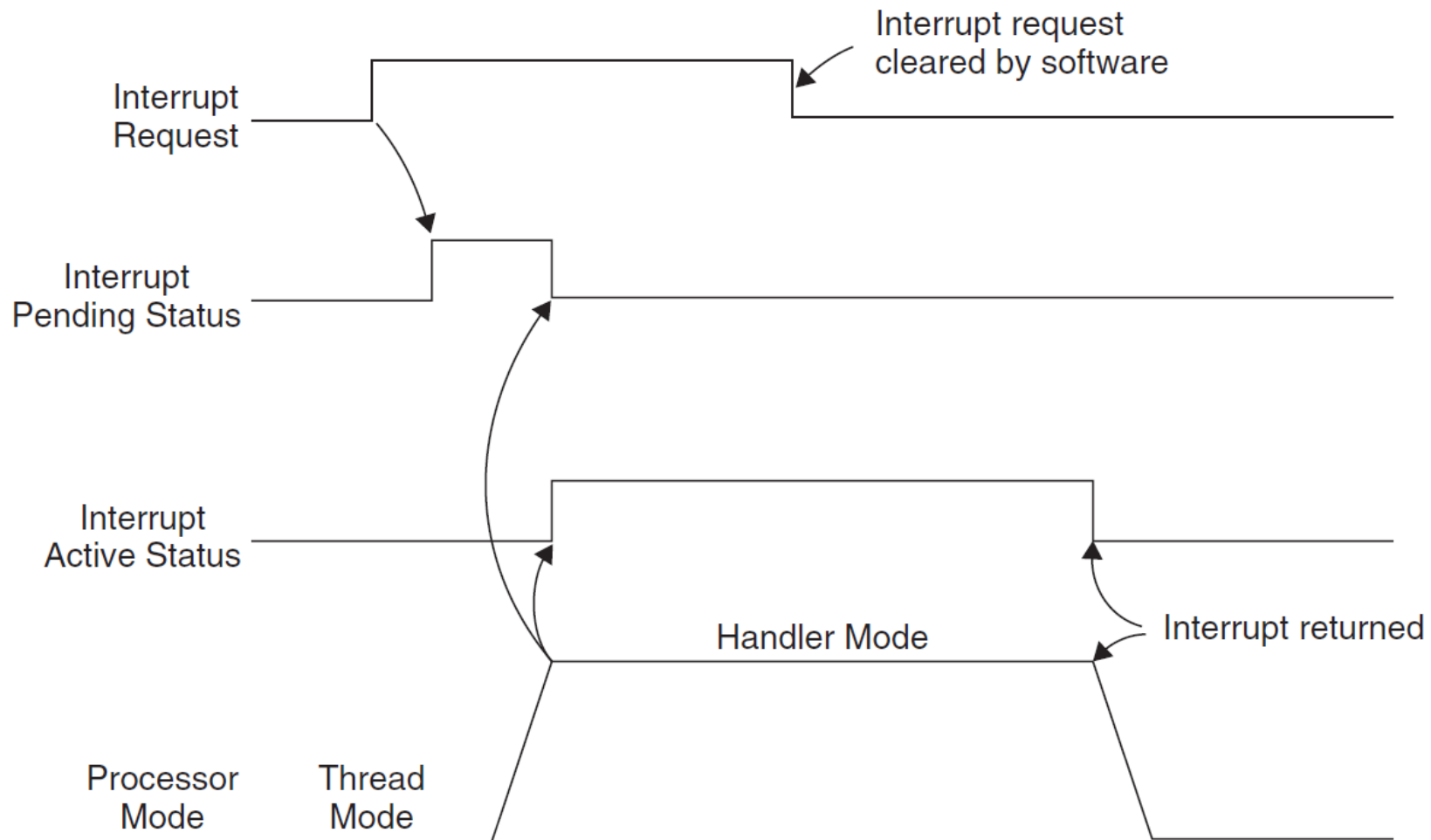
The normal case. Once Interrupt request is seen, processor puts it in “pending” state even if hardware drops the request. IPS is cleared by the hardware once we jump to the ISR.

Interrupt pending cleared before processor takes action

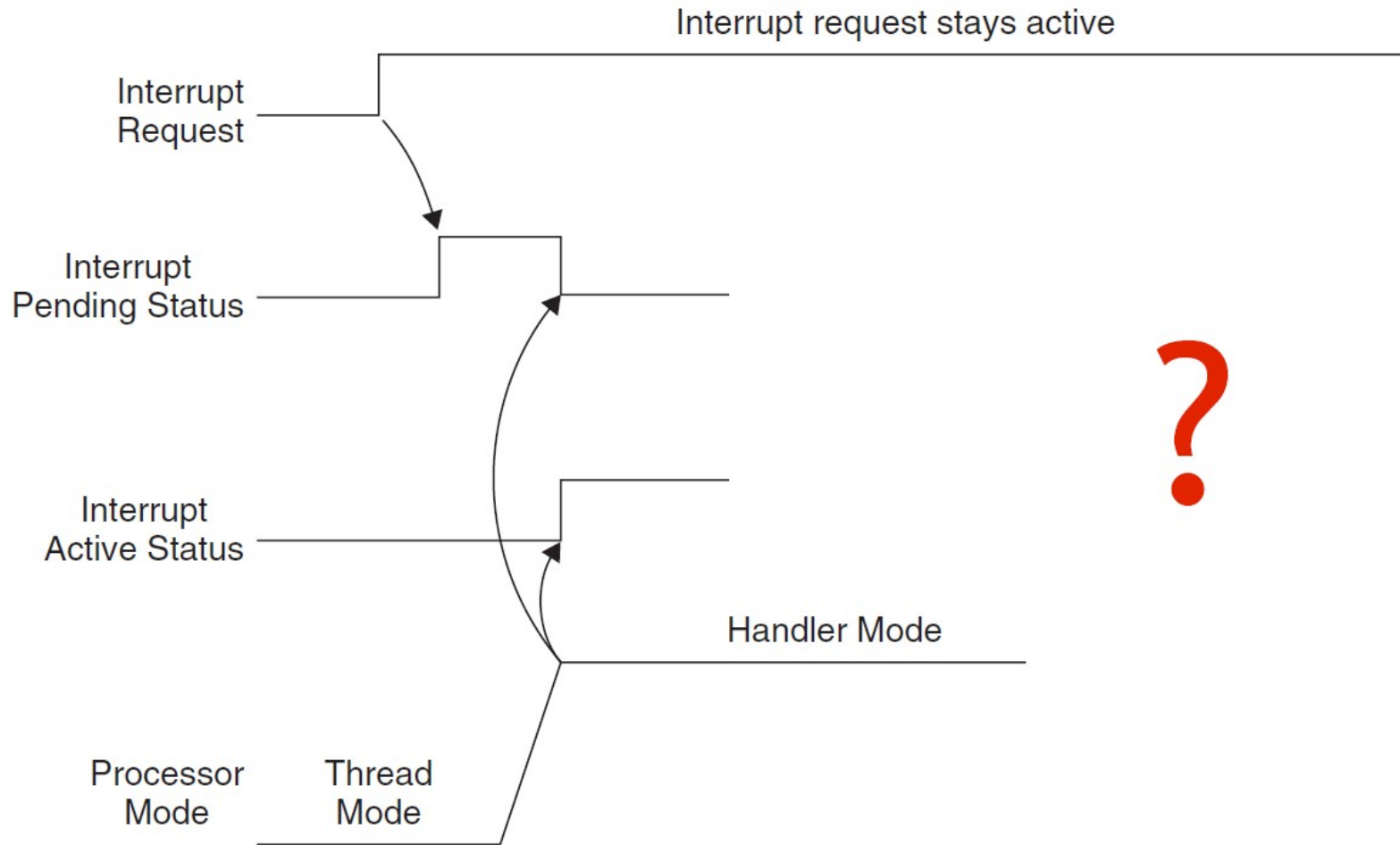


In this case, the processor never took the interrupt because we cleared the IPS by hand (via a memory-mapped I/O register)

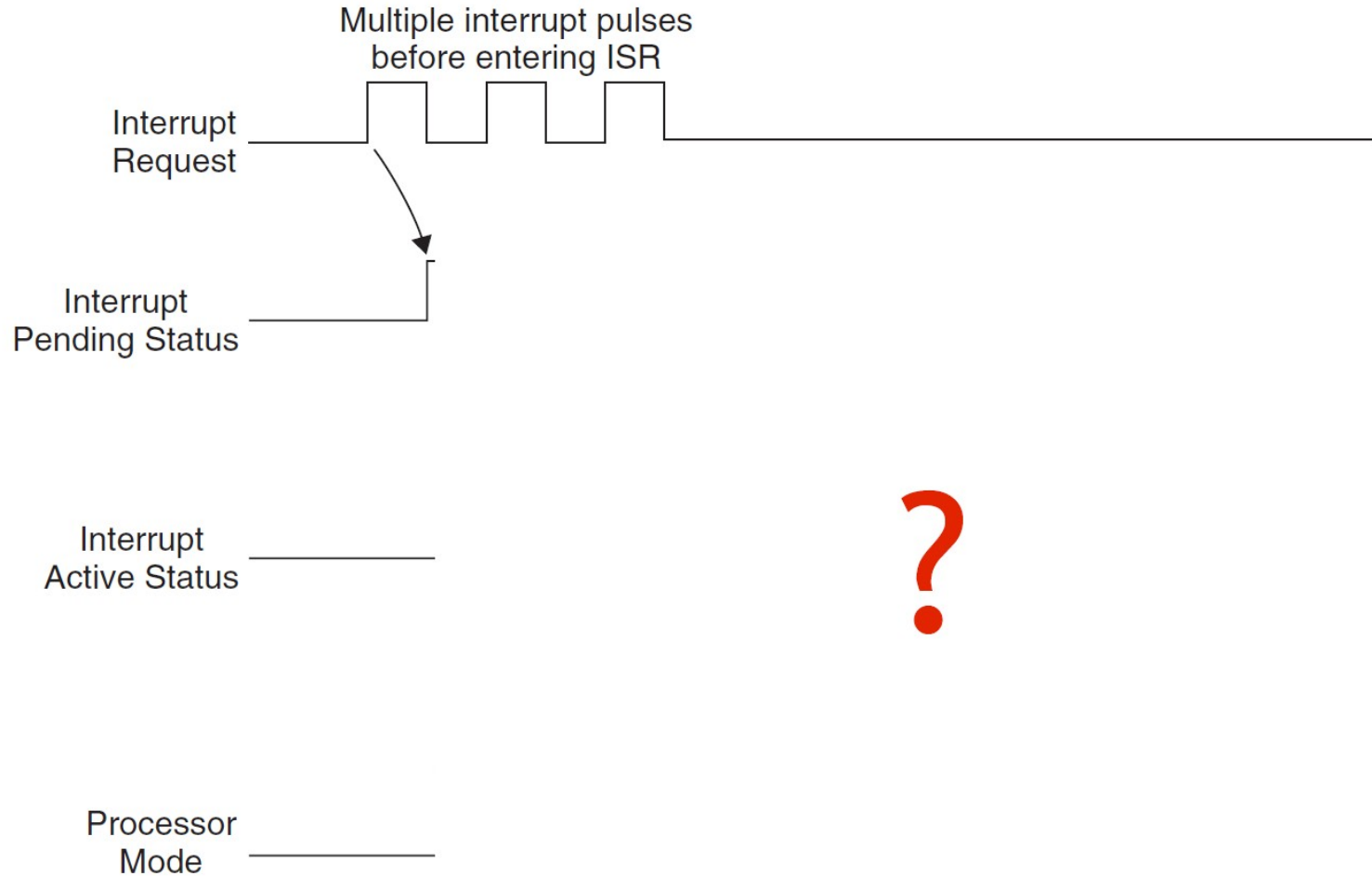
Active Status set during handler execution



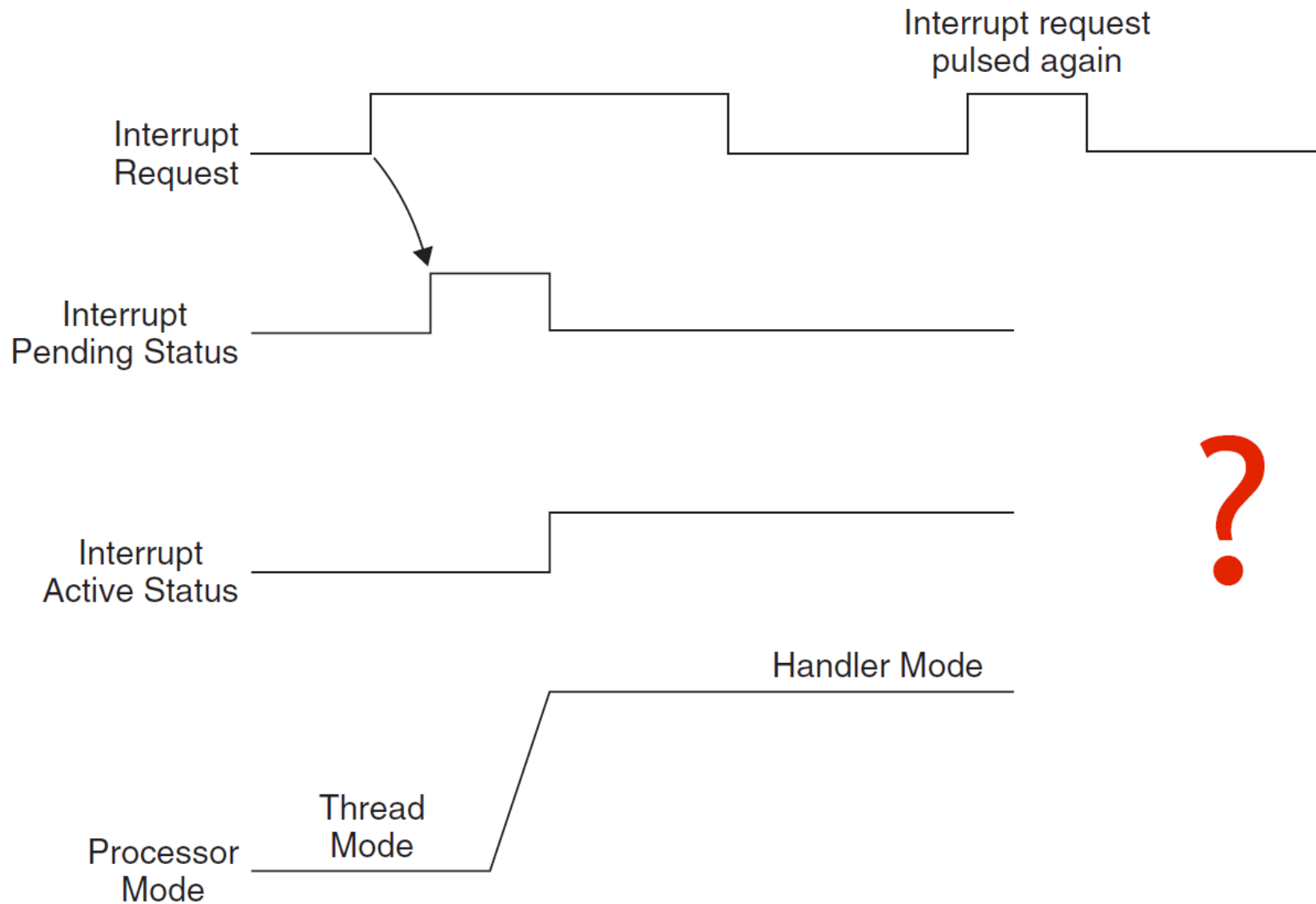
Interrupt Request not Cleared



Multiple pulses before entering interrupt



New Interrupt Request after Pending Cleared



- 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

- Set Pending & Clear Pending

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

0xE000E200	SETPEND0	R/W	0	<p>Pending for external interrupt #0–31</p> <p>bit[0] for interrupt #0 (exception #16)</p> <p>bit[1] for interrupt #1 (exception #17)</p> <p>...</p> <p>bit[31] for interrupt #31 (exception #47)</p> <p>Write 1 to set bit to 1; write 0 has no effect</p> <p>Read value indicates the current status</p>
0xE000E280	CLRPEND0	R/W	0	<p>Clear pending for external interrupt #0–31</p> <p>bit[0] for interrupt #0 (exception #16)</p> <p>bit[1] for interrupt #1 (exception #17)</p> <p>...</p> <p>bit[31] for interrupt #31 (exception #47)</p> <p>Write 1 to clear bit to 0; write 0 has no effect</p> <p>Read value indicates the current pending status</p>

- 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
...	-	-	-	-

- 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!

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* 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

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

- 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`

- 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

Address	Exception Number	Value (Word Size)
0x00000000	–	MSP initial value
0x00000004	1	Reset vector (program counter initial value)
0x00000008	2	NMI handler starting address
0x0000000C	3	Hard fault handler starting address
...	...	Other handler starting address

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

Interrupt handlers



```
--
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
--

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


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