Interrupts, Processes and Threads

Software in action: context, concurrency, synchronization

Interrupts

- An interrupt is the **automatic transfer of software execution** in response to a **hardware event** that is asynchronous with the current software execution
- This hardware event is called a **trigger** and it breaks the execution flow of the main thread of the program
- The event 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

Possible hardware events

- A busy-to-ready transition in an external I/O device
  - Peripheral/device, e.g., UART input/output device
  - Reset button, Timer expires, Power failure, System error
- An internal event
  - Bus fault, memory fault
  - A periodic timer
  - Div. by zero, illegal/unsupported instruction

When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag

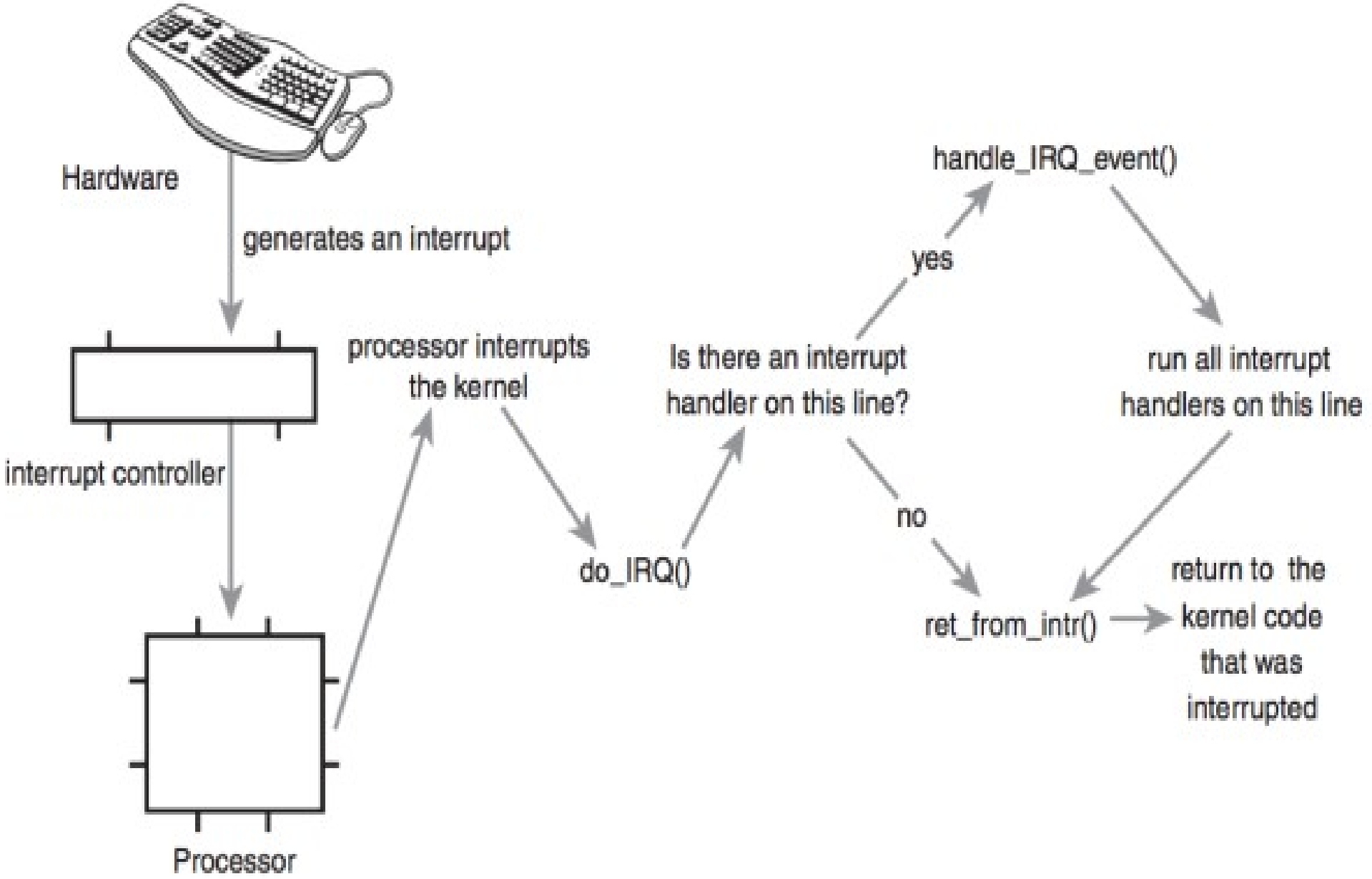
Example: Cortex M3 Interrupts

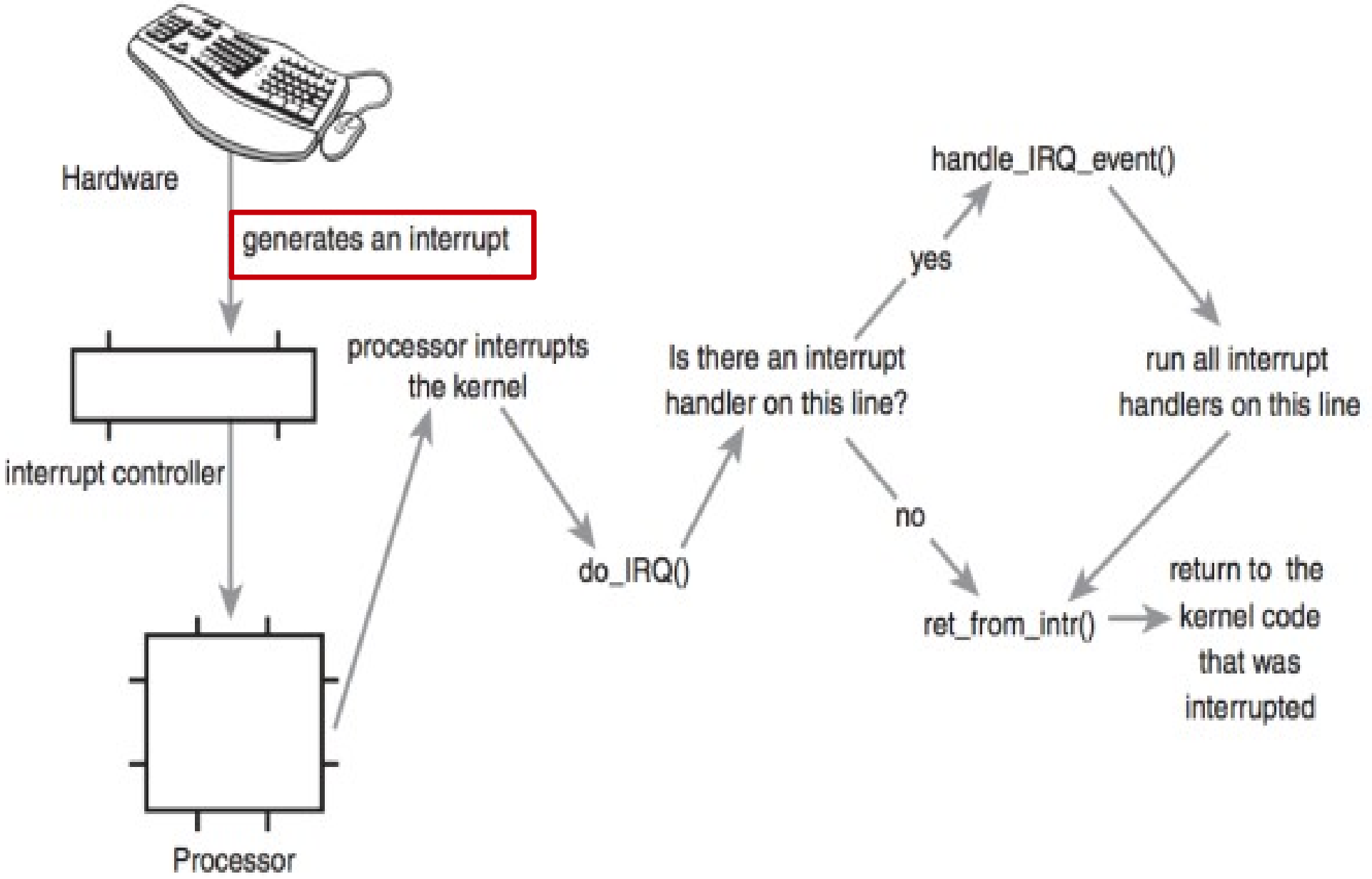
- Exceptions
  - System exceptions: numbered 1 to 15
  - External interrupt inputs: numbered from 16 up
- Different numbers of external interrupt inputs (from 1 to 240) and different numbers of priority levels
- Value of the current running exception is indicated by
  - The special register Interrupt Program Status Register (IPSR)
  - From the NVIC's Interrupt Control State Register (the VECTACTIVE field)

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; MPU violation or access to illegal locations
5	Bus Fault	Programmable	Bus error; occurs when 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 due to program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	–
11	SVCall	Programmable	System Service call
12	Debug Monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	–
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System Tick Timer

External Interrupts

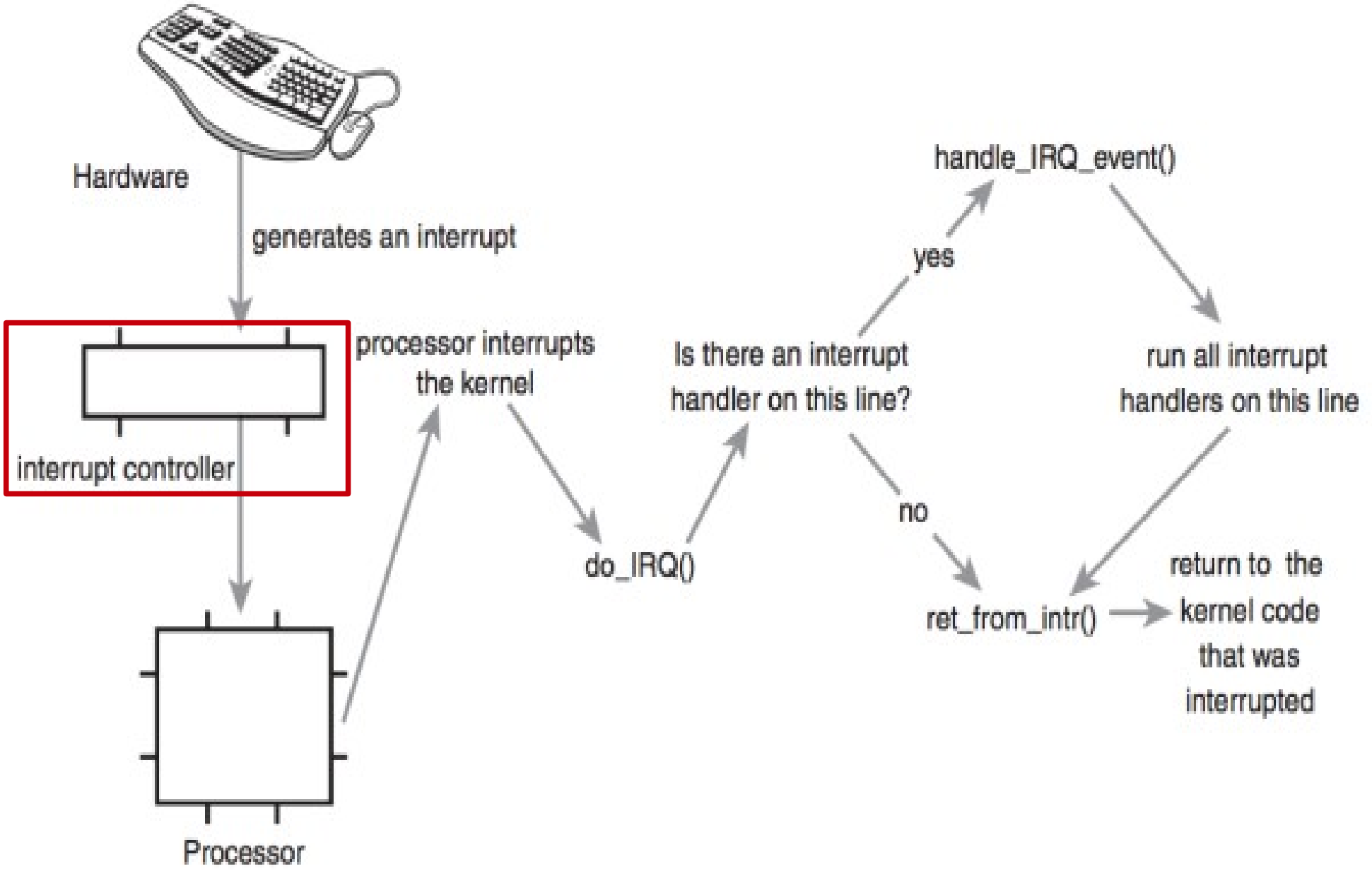



Hardware
generates an interrupt
interrupt controller
processor interrupts
the kernel
Processor
do_IRQ()
Is there an interrupt
handler on this line?
yes
handle_IRQ_event()
run all interrupt
handlers on this line
no
ret_from_intr()
return to the
kernel code
that was
interrupted

Interrupt Programming

- To **arm (disarm)** a device/peripheral means to enable (shut off) the source of interrupts. Each potential interrupting trigger has a separate “arm” bit. One arms (disarms) a trigger if one is (is not) interested in interrupts from this source.

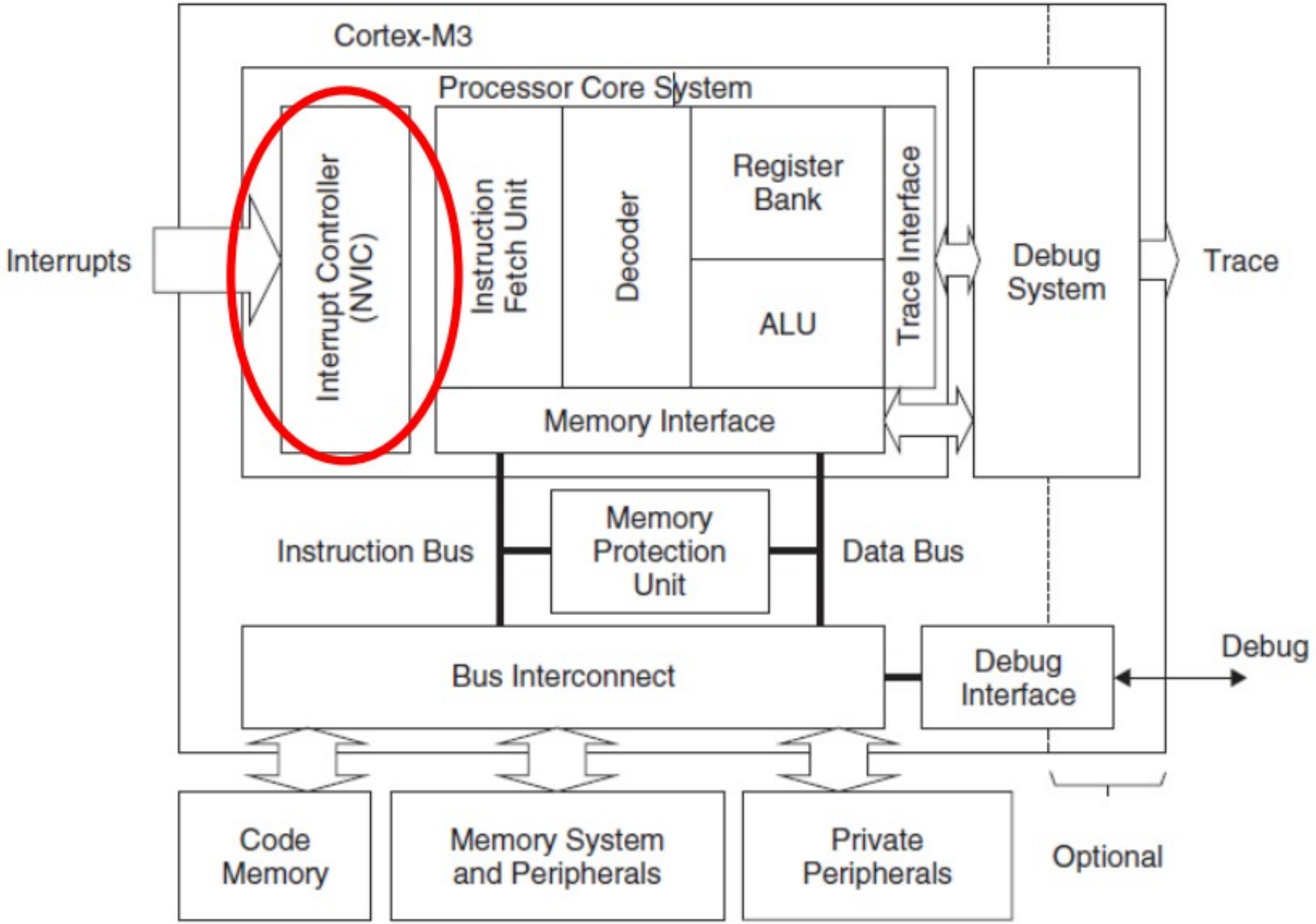
- To **enable (disable)** means to allow interrupts at this time (postponing interrupts until a later time). On the ARM Coretx-M3 processor, there is one interrupt enable bit for the entire interrupt system. In particular, to disable interrupts we set the interrupt mask bit, I, in PRIMASK register.


Hardware
generates an interrupt
interrupt controller
processor interrupts
the kernel
Processor
do_IRQ()
Is there an interrupt
handler on this line?
yes
handle_IRQ_event()
run all interrupt
handlers on this line
no
ret_from_intr()
return to the
kernel code
that was
interrupted

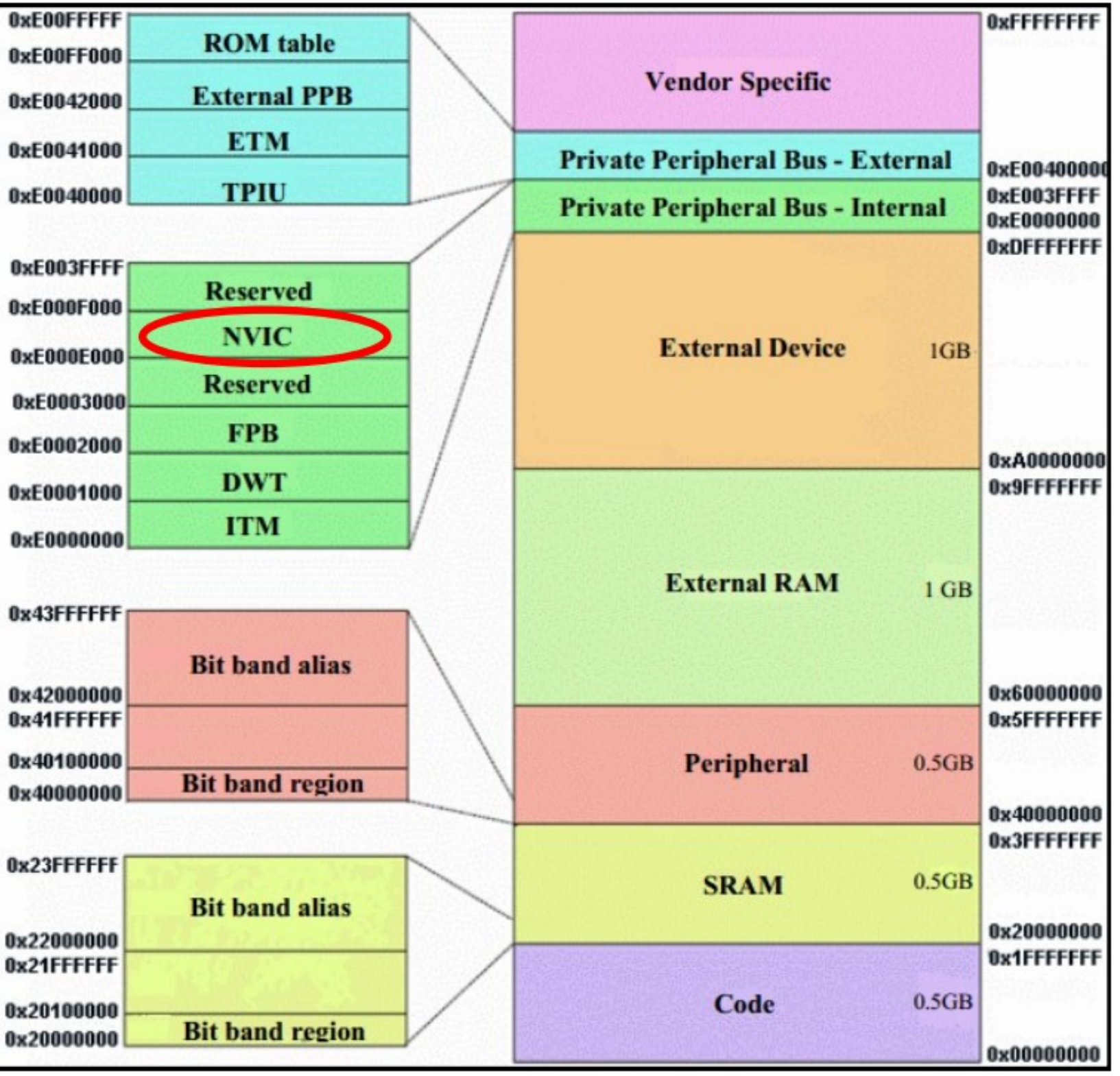
Nested Vectored Interrupt Controller (NVIC)

- Interrupts on the Cortex-M3 are controlled by the Nested Vector Interrupt Controller (NVIC)
- NVIC supports 1 to 240 external interrupt inputs

  (commonly known as IRQs)
- NVIC control registers are accessible as memory-mapped devices
- NVIC can be accessed as memory location 0xE000E000


Cortex-M3
Processor Core System
Interrupts
Interrupt Controller (NVIC)
Instruction Fetch Unit
Decoder
Register Bank
ALU
Trace Interface
Debug System
Trace
Memory Interface
Instruction Bus
Memory Protection Unit
Data Bus
Bus Interconnect
Debug Interface
Debug
Code Memory
Memory System and Peripherals
Private Peripherals
Optional

Memory Map

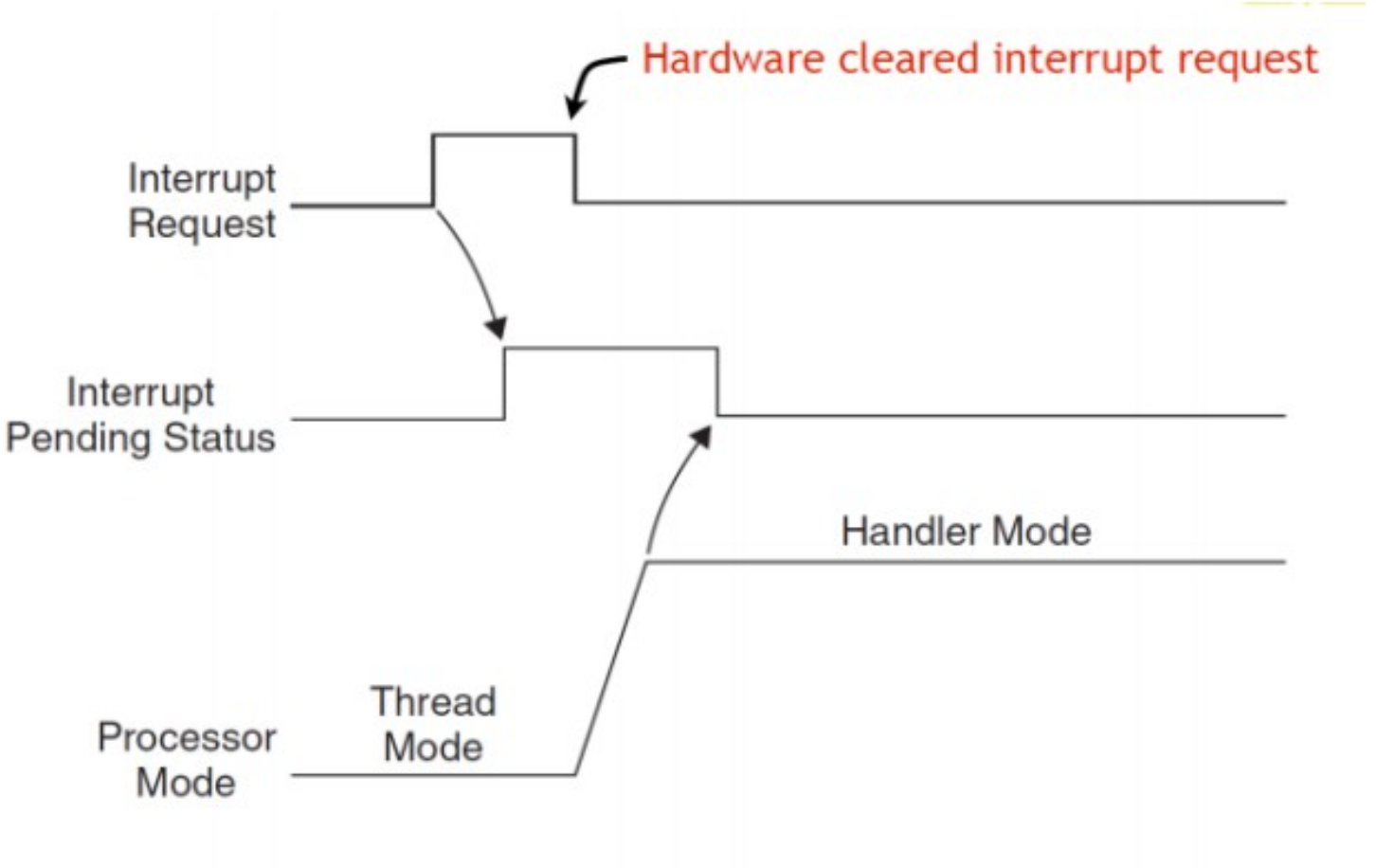

0xE00FFFFF
ROM table
0xE00FF000
External PPB
0xE0042000
ETM
0xE0041000
TPIU
0xE0040000
0xE003FFFF
Reserved
0xE000F000
NVIC
0xE000E000
Reserved
0xE0003000
FPB
0xE0002000
DWT
0xE0001000
ITM
0xE0000000
0x43FFFFFF
Bit band alias
0x42000000
0x41FFFFFF
0x40100000
Bit band region
0x40000000
0x23FFFFFF
Bit band alias
0x22000000
0x21FFFFFF
0x20100000
Bit band region
0x20000000
0xFFFFFFFF
Vendor Specific
Private Peripheral Bus - External
0xE00400000
0xE003FFFF
Private Peripheral Bus - Internal
0xE0000000
0xDFFFFFFF
External Device
1GB
0xA0000000
0x9FFFFFFF
External RAM
1 GB
0x60000000
0x5FFFFFFF
Peripheral
0.5GB
0x40000000
0x3FFFFFFF
SRAM
0.5GB
0x20000000
0x1FFFFFFF
Code
0.5GB
0x00000000

NVIC registers

Table 6-1 NVIC registers

Address	Name	Type	Reset	Description
0xE000E004	ICTR	RO	-	*Interrupt Controller Type Register, ICTR*
0xE000E100 - 0xE000E11C	NVIC_ISER0 - NVIC_ISER7	RW	0x00000000	Interrupt Set-Enable Registers
0xE000E180 - 0E000xE19C	NVIC_ICER0 - NVIC_ICER7	RW	0x00000000	Interrupt Clear-Enable Registers
0xE000E200 - 0xE000E21C	NVIC_ISPR0 - NVIC_ISPR7	RW	0x00000000	Interrupt Set-Pending Registers
0xE000E280 - 0xE000E29C	NVIC_ICPR0 - NVIC_ICPR7	RW	0x00000000	Interrupt Clear-Pending Registers
0xE000E300 - 0xE000E31C	NVIC_IABR0 - NVIC_IABR7	RO	0x00000000	Interrupt Active Bit Register
0xE000E400 -	NVIC_IPR0 -	RW	0x00000000	Interrupt Priority Register

Default behavior

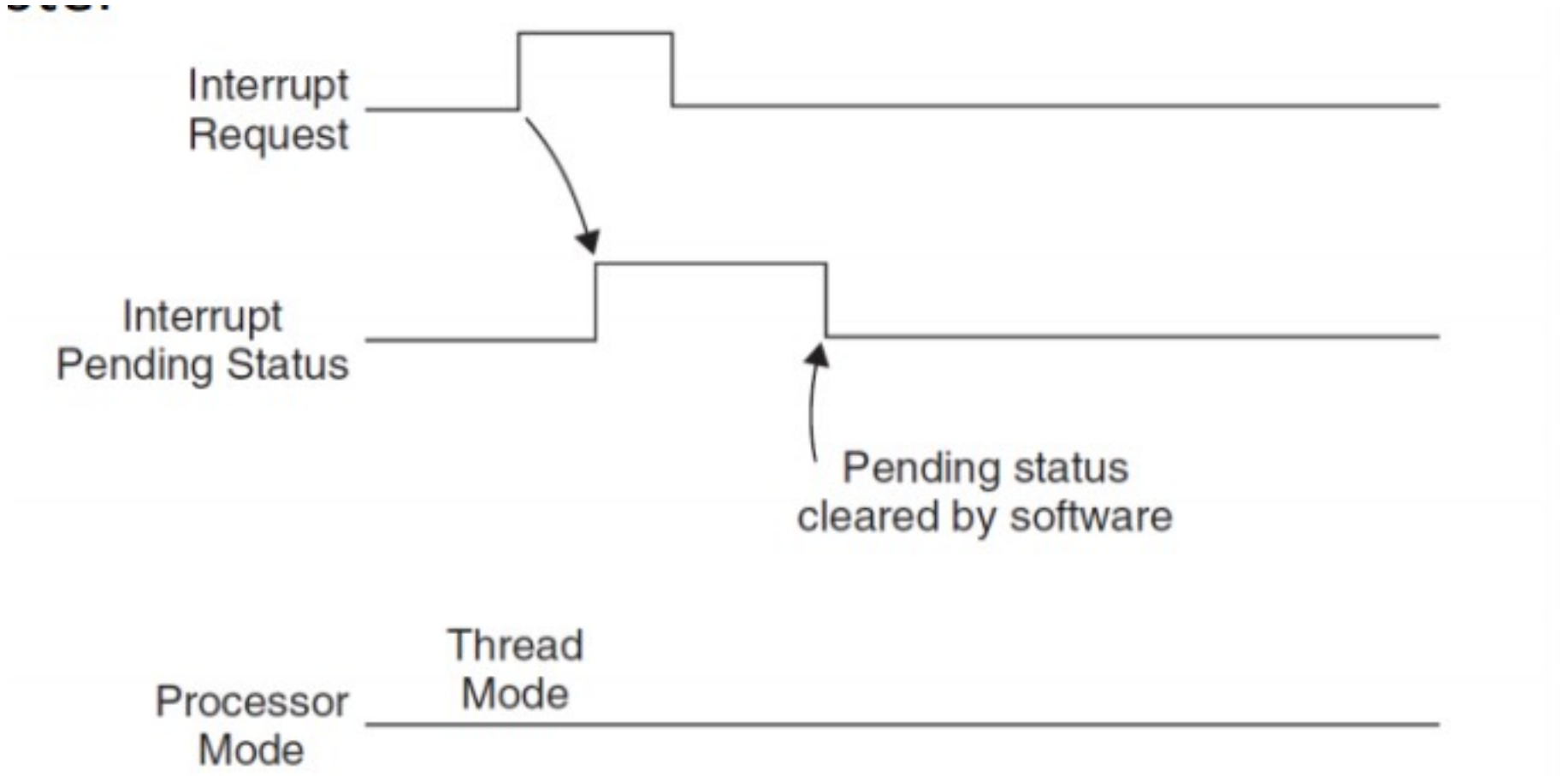


Use NVIC registers

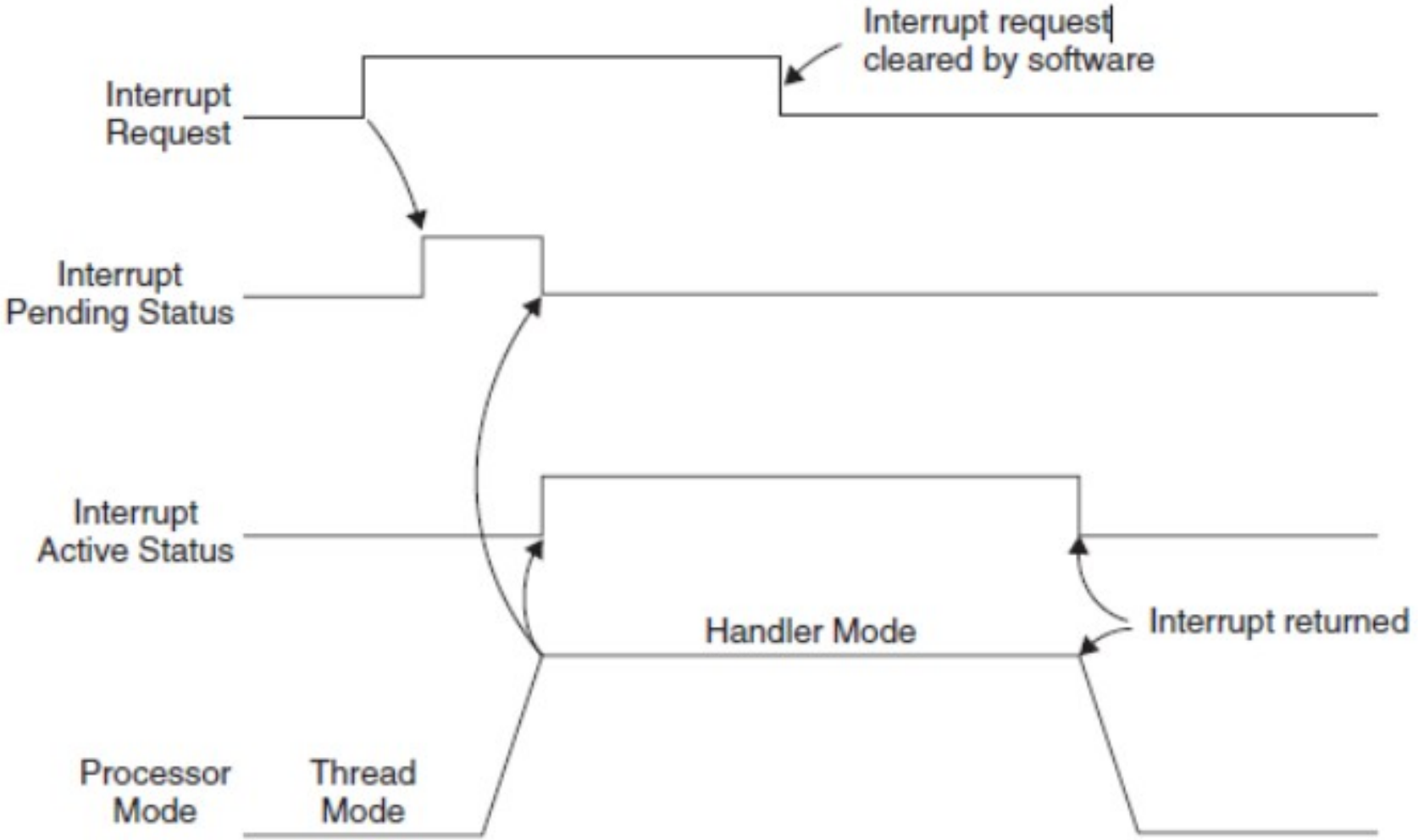
Table 6-1 NVIC registers

Address	Name	Type	Reset	Description
0xE000E004	ICTR	RO	-	*Interrupt Controller Type Register, ICTR*
0xE000E100 - 0xE000E11C	NVIC_ISER0 - NVIC_ISER7	RW	0x00000000	Interrupt Set-Enable Registers
0xE000E180 - 0E000xE19C	NVIC_ICER0 - NVIC_ICER7	RW	0x00000000	Interrupt Clear-Enable Registers
0xE000E200 - 0xE000E21C	NVIC_ISPR0 - NVIC_ISPR7	RW	0x00000000	Interrupt Set-Pending Registers
0xE000E280 - 0xE000E29C	NVIC_ICPR0 - NVIC_ICPR7	RW	0x00000000	Interrupt Clear-Pending Registers
0xE000E300 - 0xE000E31C	NVIC_IABR0 - NVIC_IABR7	RO	0x00000000	Interrupt Active Bit Register
0xE000E400 -	NVIC_IPR0 -	RW	0x00000000	Interrupt Priority Register

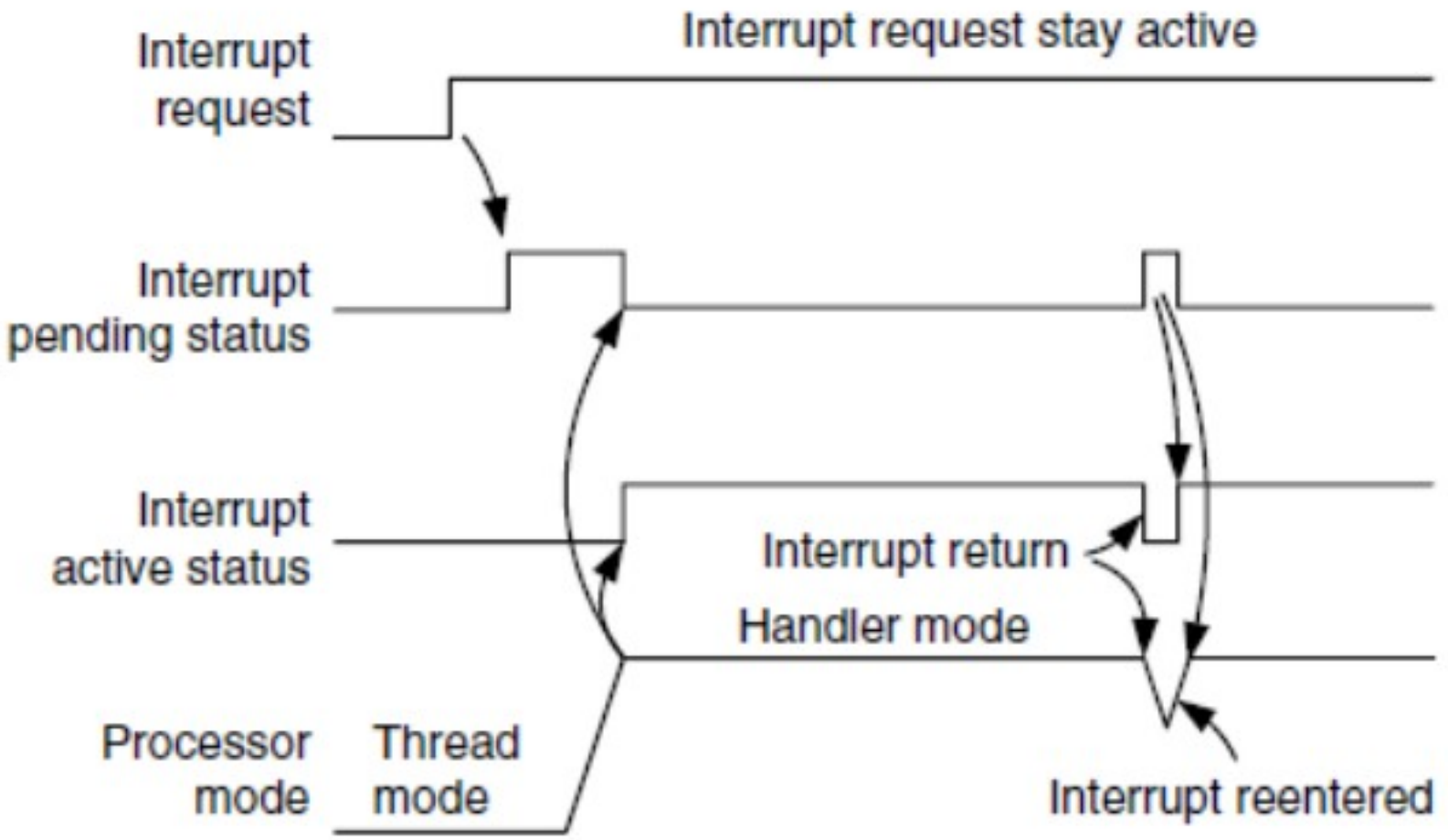
To Ignore Interrupts



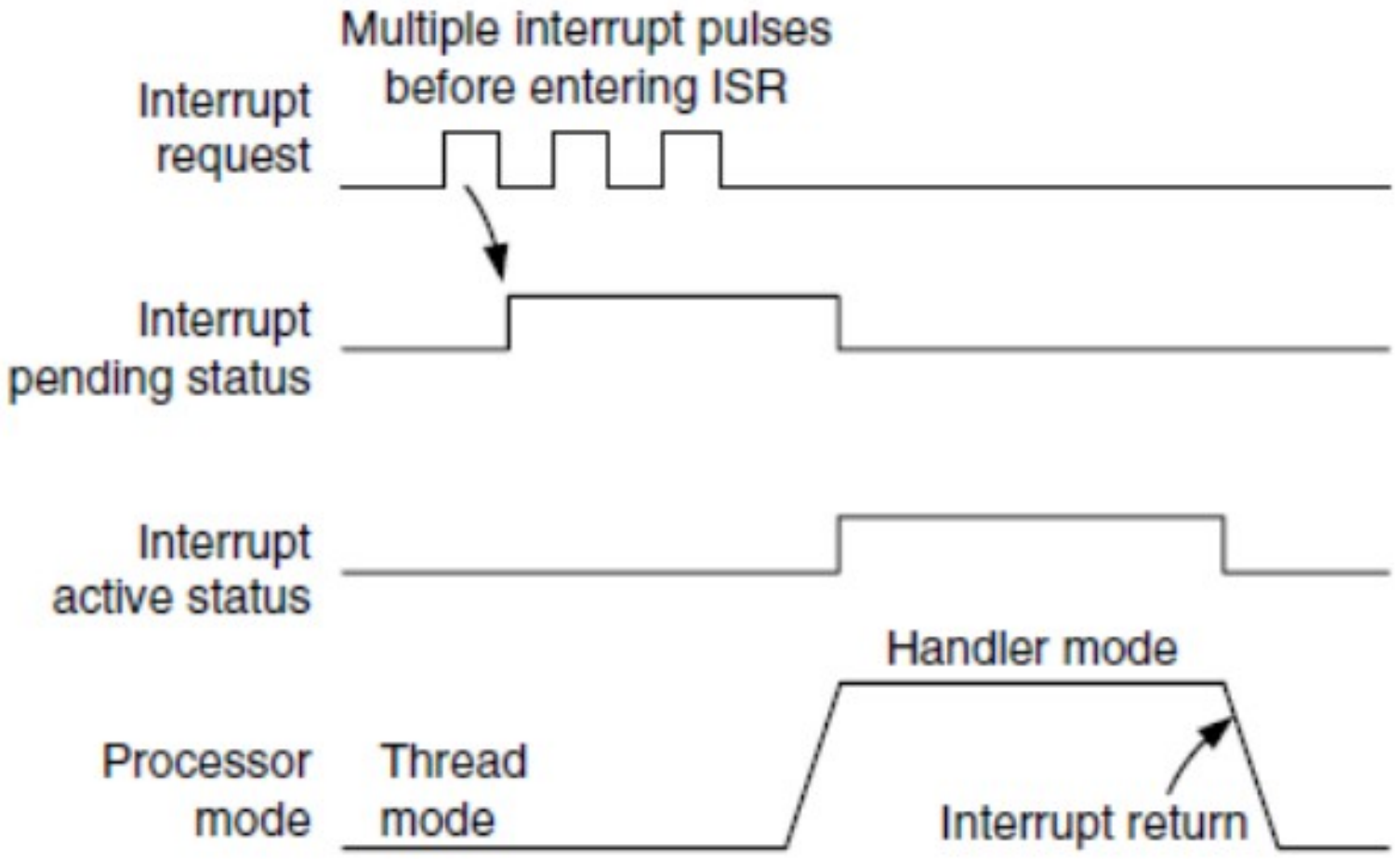
Interrupt Request cleared in software



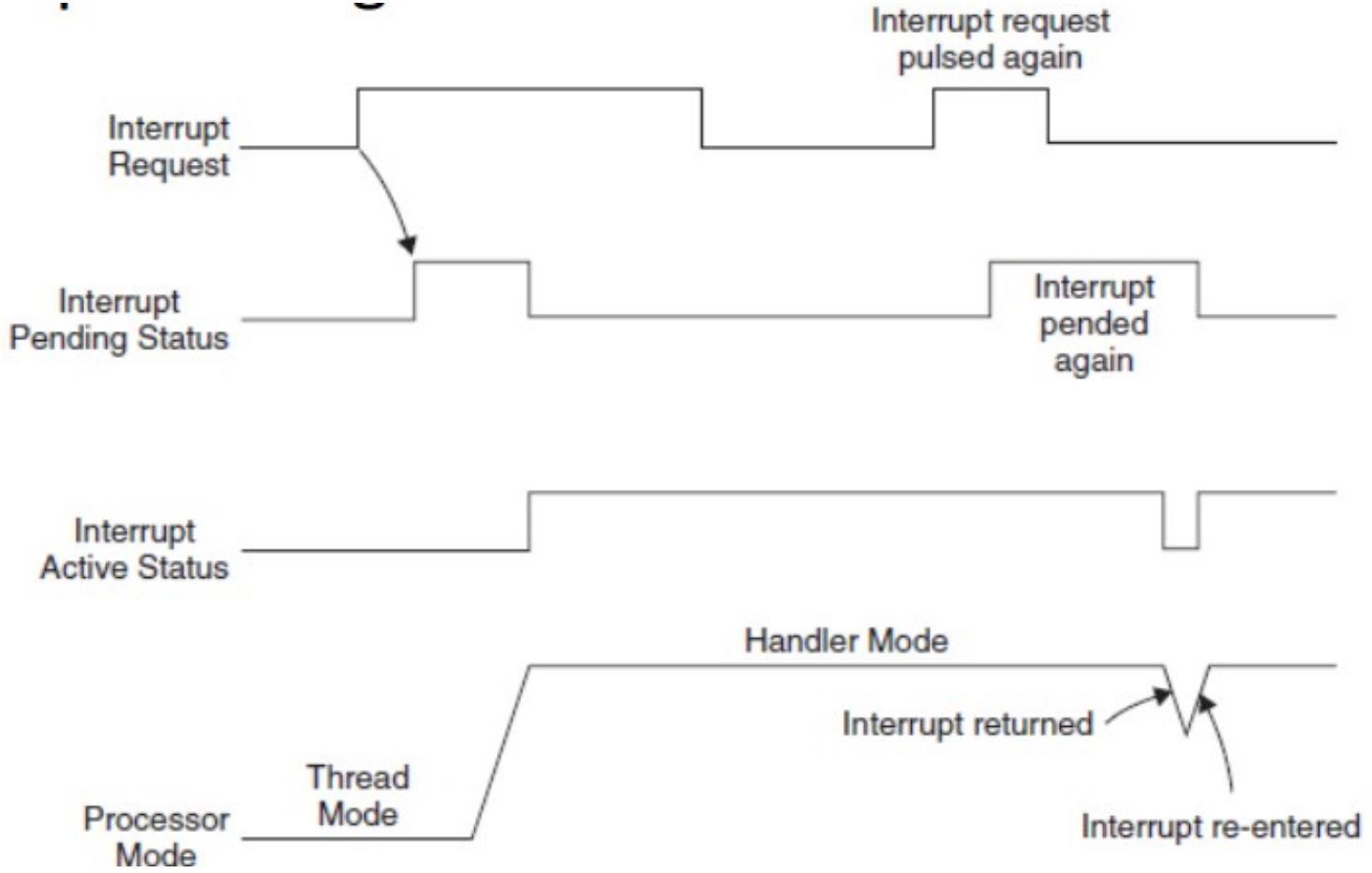
Otherwise, handles an already processed interrupt

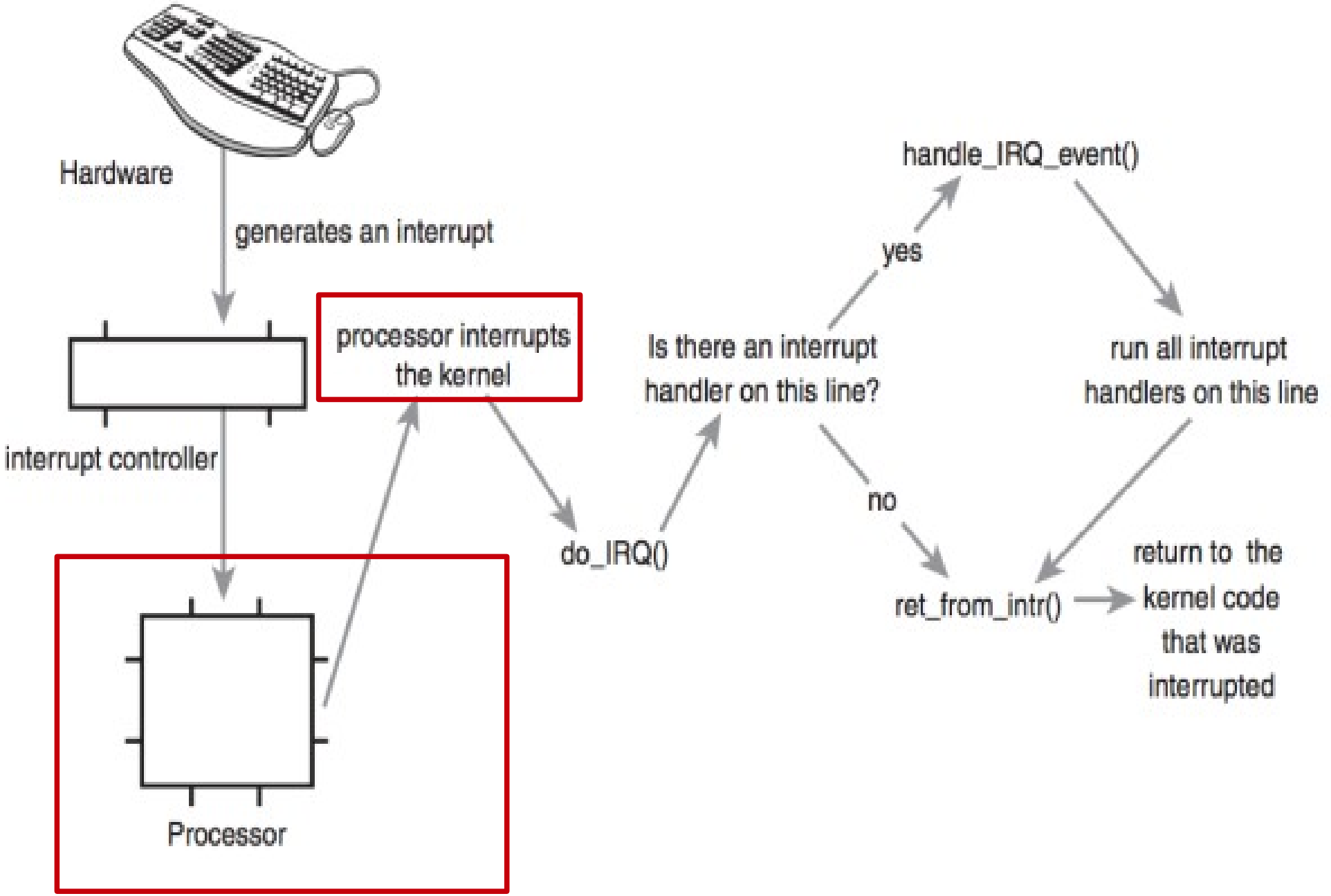


High Interrupt Frequency



Can still be processed, if recorded




Hardware
generates an interrupt
interrupt controller
processor interrupts the kernel
Processor
do_IRQ()
Is there an interrupt handler on this line?
yes
handle_IRQ_event()
run all interrupt handlers on this line
no
ret_from_intr()
return to the kernel code that was interrupted

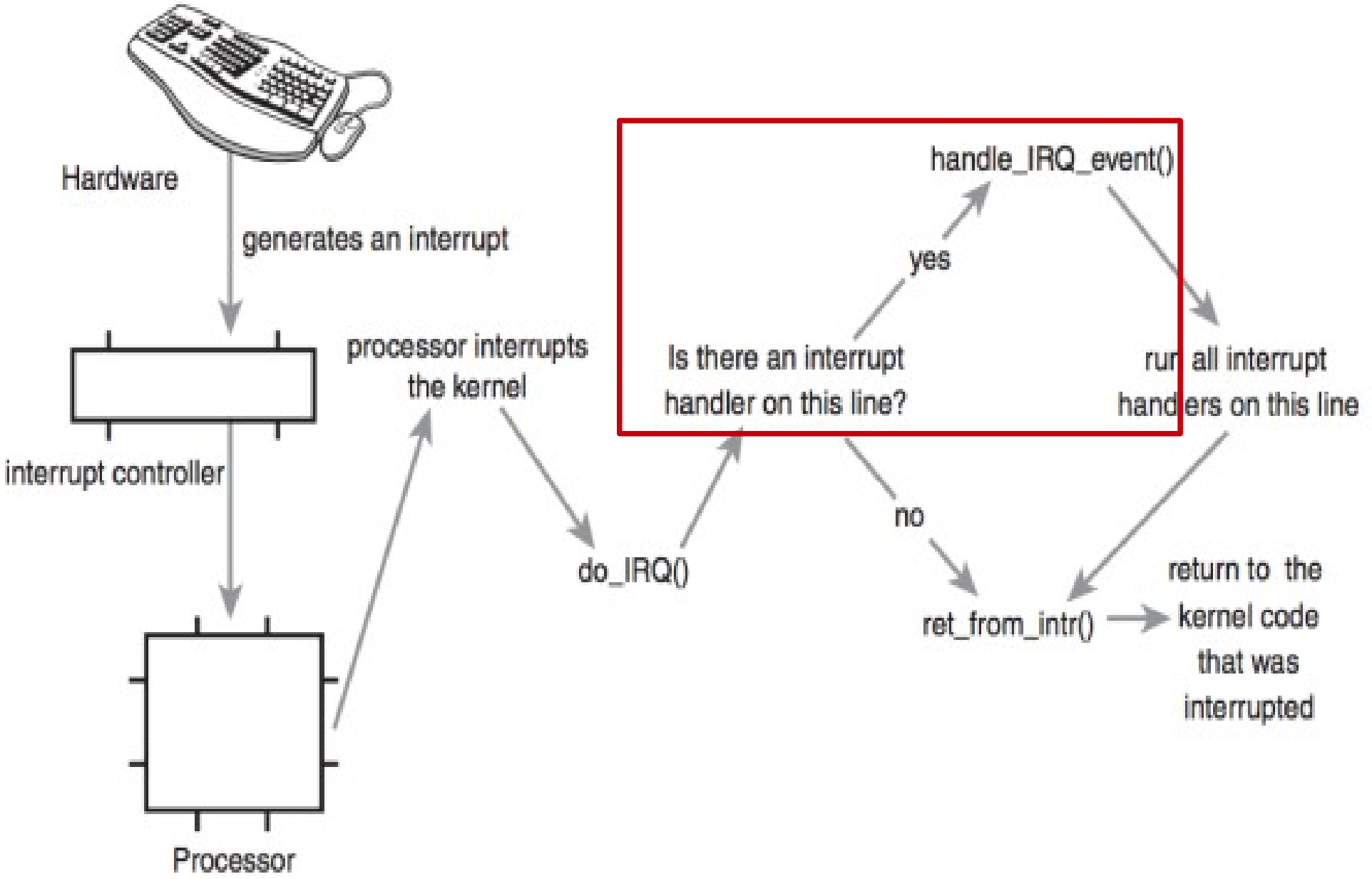
Vector Tables

- When an exception takes place and is being handled by the Cortex-M3, the processor will need to locate the starting address of the exception handler
- This information is stored in the vector table
- Each exception has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located
- Vectors are stored in ROM at the beginning of the memory

Exception Vector Table after powerup

- Exception Vector Table after power up is located at address 0x00000000
- ROM location 0x00000000 has the initial stack pointer
- Location 0x00000004 contains the initial program counter (pc) called the reset vector
- Reset vector points to a function called reset handler, which is the first thing executed following reset
- Vector table can be relocated to change interrupt handlers at runtime (vector table offset register)

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


Hardware
generates an interrupt
interrupt controller
Processor
processor interrupts
the kernel
do_IRQ()
Is there an interrupt
handler on this line?
yes
handle_IRQ_event()
run all interrupt
handlers on this line
no
ret_from_intr()
return to the
kernel code
that was
interrupted

Interrupt Handler

An interrupt handler or interrupt service routine (ISR) is the function that the kernel runs in response to a specific interrupt

- Each device that generates interrupts has an associated interrupt handler.
- The interrupt handler for a device is part of the device's driver (the kernel code that manages the device)
- Each device has one associated driver. If that device uses interrupts (and most do), that driver must register one interrupt handler.

Linux Interrupt Handlers

- Interrupt handlers are normal C functions
  - Drivers can register an interrupt handler and enable a given interrupt line for handling with the function request_irq(), which is declared in <linux/interrupt.h>
  - The first parameter, irq, specifies the interrupt number to allocate
    - For some devices (e.g. legacy PC devices such as the system timer or keyboard), this value is typically hard-coded.
    - For most other devices, it is probed or otherwise determined programmatically and dynamically.
  - The second parameter, handler, is a function pointer to the actual interrupt handler that services this interrupt.

```
/* request_irq: allocate a given interrupt line */
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev)
```

Example RTC Handler Function
drivers/char/rtc.c

```
static irqreturn_t rtc_interrupt(int irq, void *dev_id)
{
    /*
     *  Can be an alarm interrupt, update complete interrupt,
     *  or a periodic interrupt. We store the status in the
     *  low byte and the number of interrupts received since
     *  the last read in the remainder of rtc_irq_data.
     */

    spin_lock(&rtc_lock);
    rtc_irq_data += 0x100;
    rtc_irq_data &= ~0xff;
    if (is_hpet_enabled()) {
        /*
         * In this case it is HPET RTC interrupt handler
         * calling us, with the interrupt information
         * passed as arg1, instead of irq.
         */
        rtc_irq_data |= (unsigned long)irq & 0xF0;
    } else {
        rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);
    }

    if (rtc_status & RTC_TIMER_ON)
        mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);

    spin_unlock(&rtc_lock);

    /* Now do the rest of the actions */
    spin_lock(&rtc_task_lock);
    if (rtc_callback)
        rtc_callback->func(rtc_callback->private_data);
    spin_unlock(&rtc_task_lock);
    wake_up_interruptible(&rtc_wait);

    kill_fasync(&rtc_async_queue, SIGIO, POLL_IN);

    return IRQ_HANDLED;
}
```

Interrupt Context

Interrupt Handlers run in a special **context** called interrupt context or atomic context.

Code executing from interrupt context cannot do the following

- Go to sleep or relinquish the processor
- Acquire a mutex
- Perform time-consuming tasks
- Access user space virtual memory