Interrupts

Interrupts

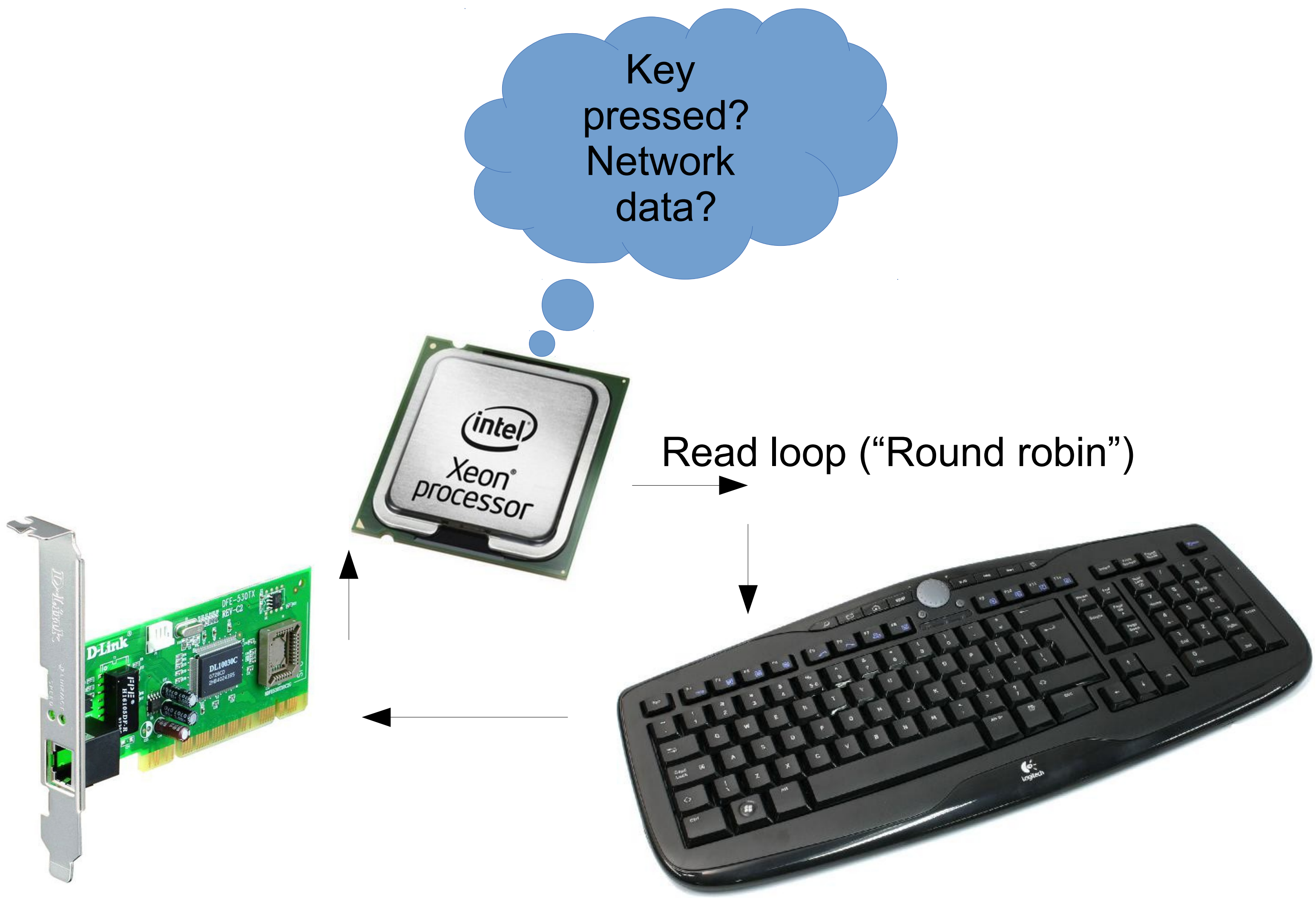
“Polling”
versus
Interrupts

Interrupts

Polling

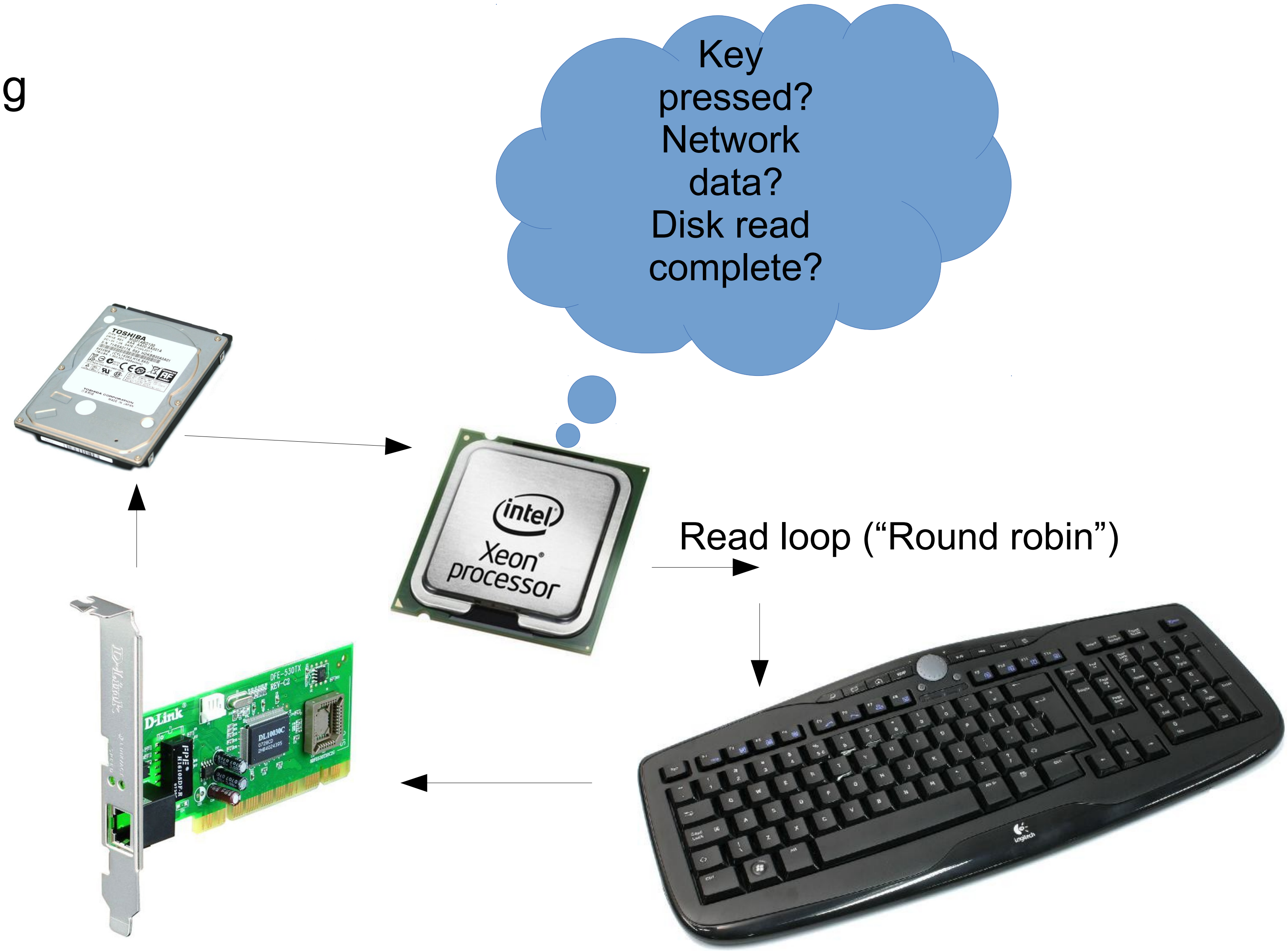
Interrupts

Polling



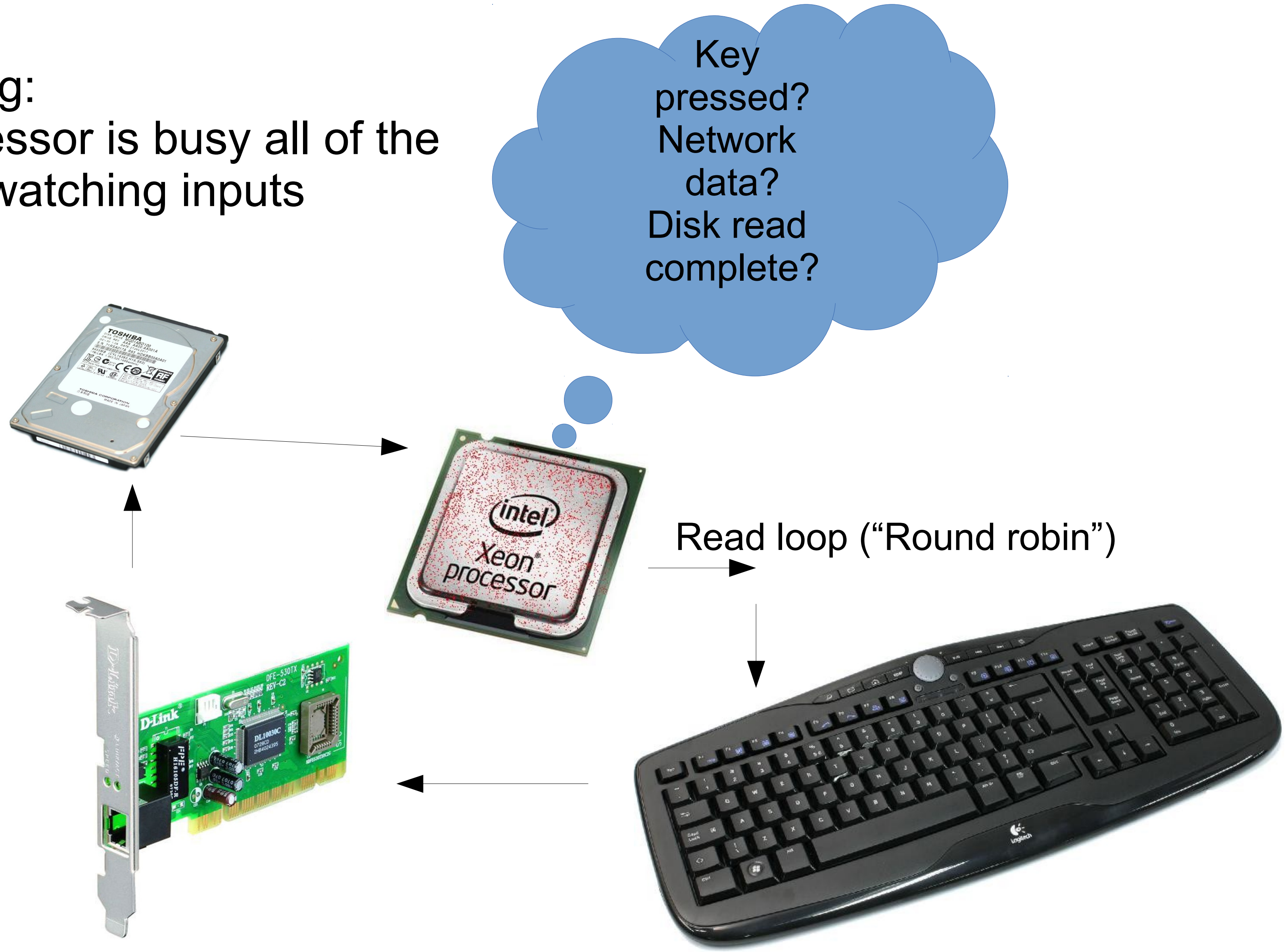
Interrupts

Polling



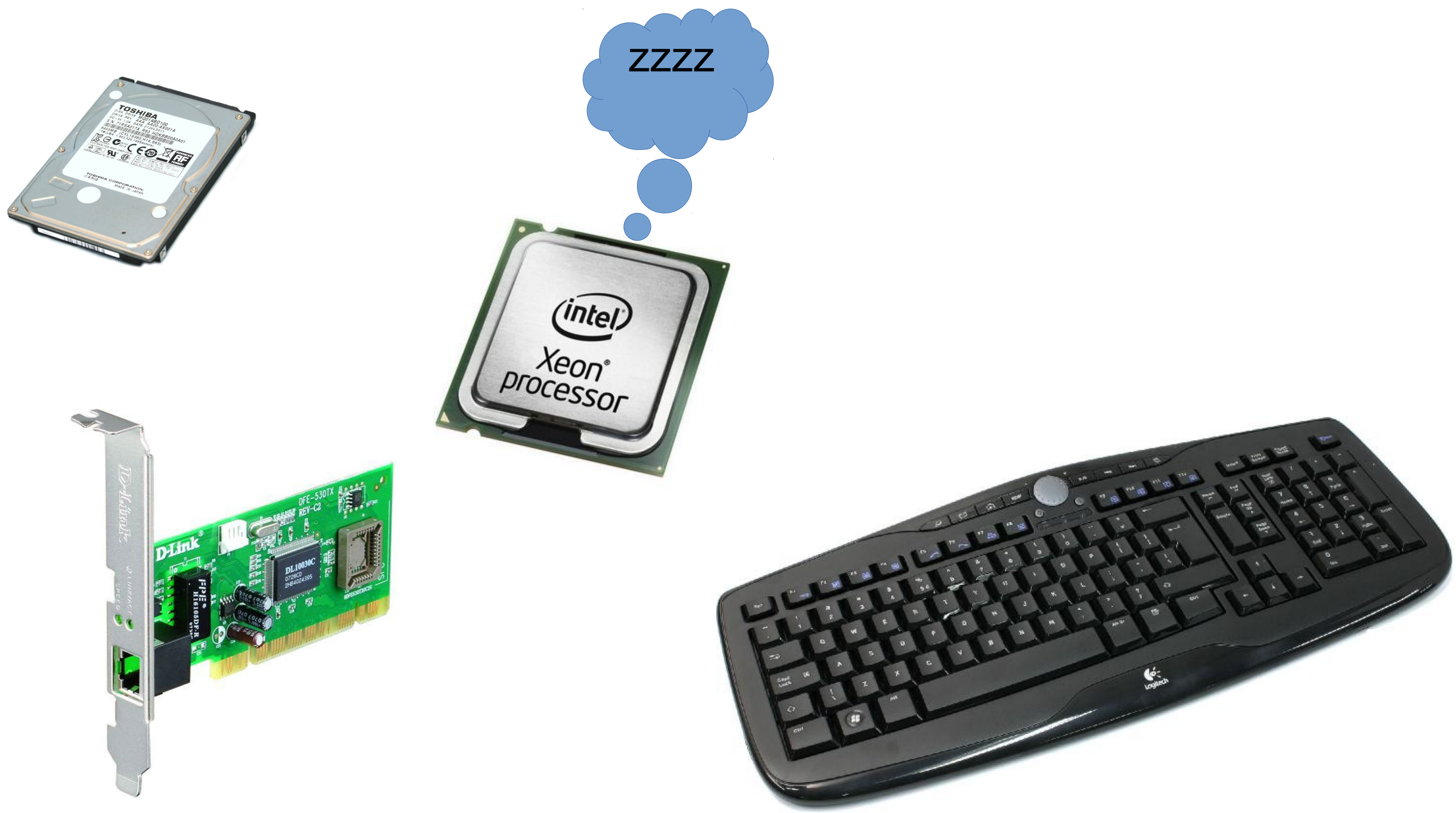
Interrupts

Polling:
Processor is busy all of the time watching inputs



Interrupts

Interrupts: Processor idle for extended periods

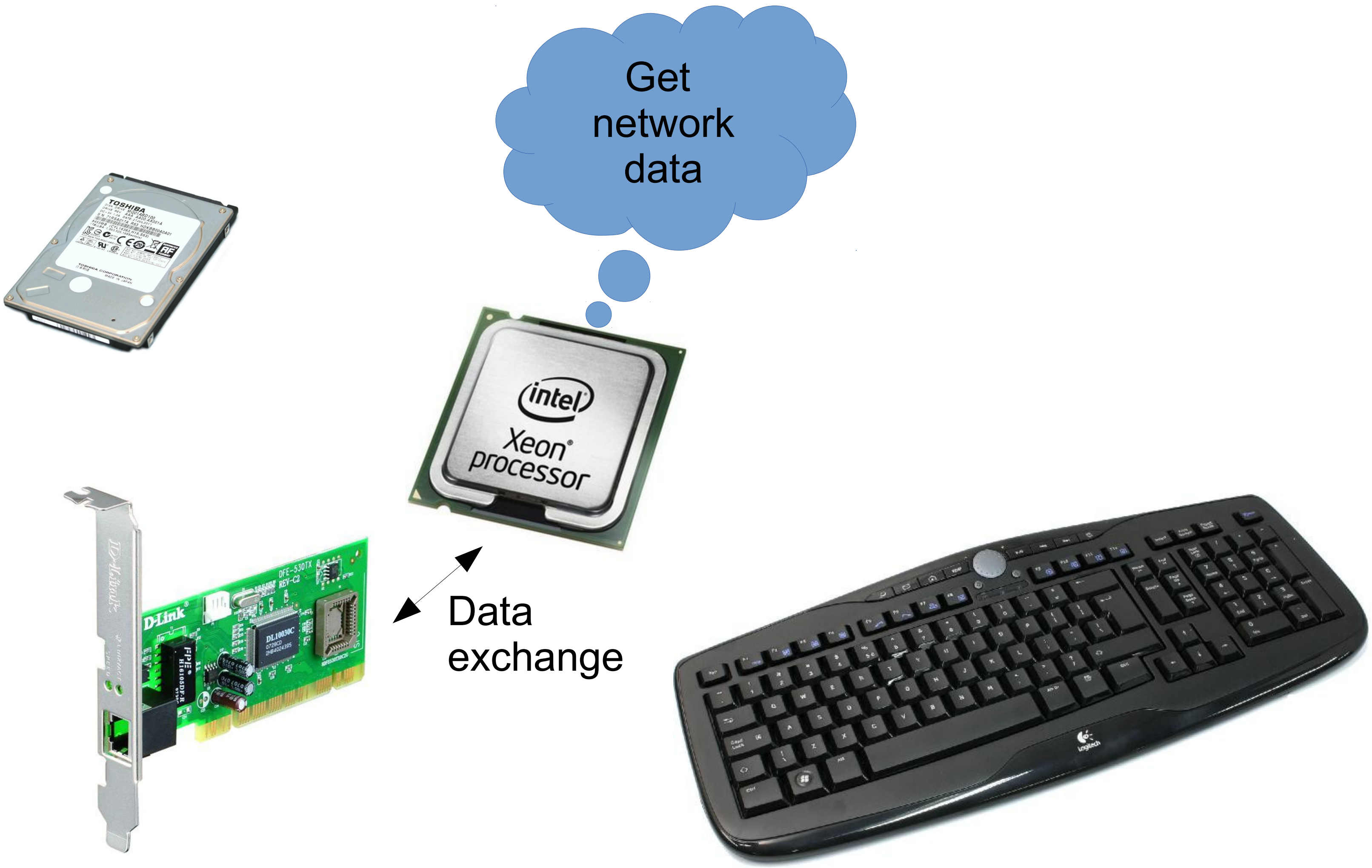


Interrupts

Interrupts: Device needing attention raises Interrupt ReQuest:
IRQ (a hardware signal)

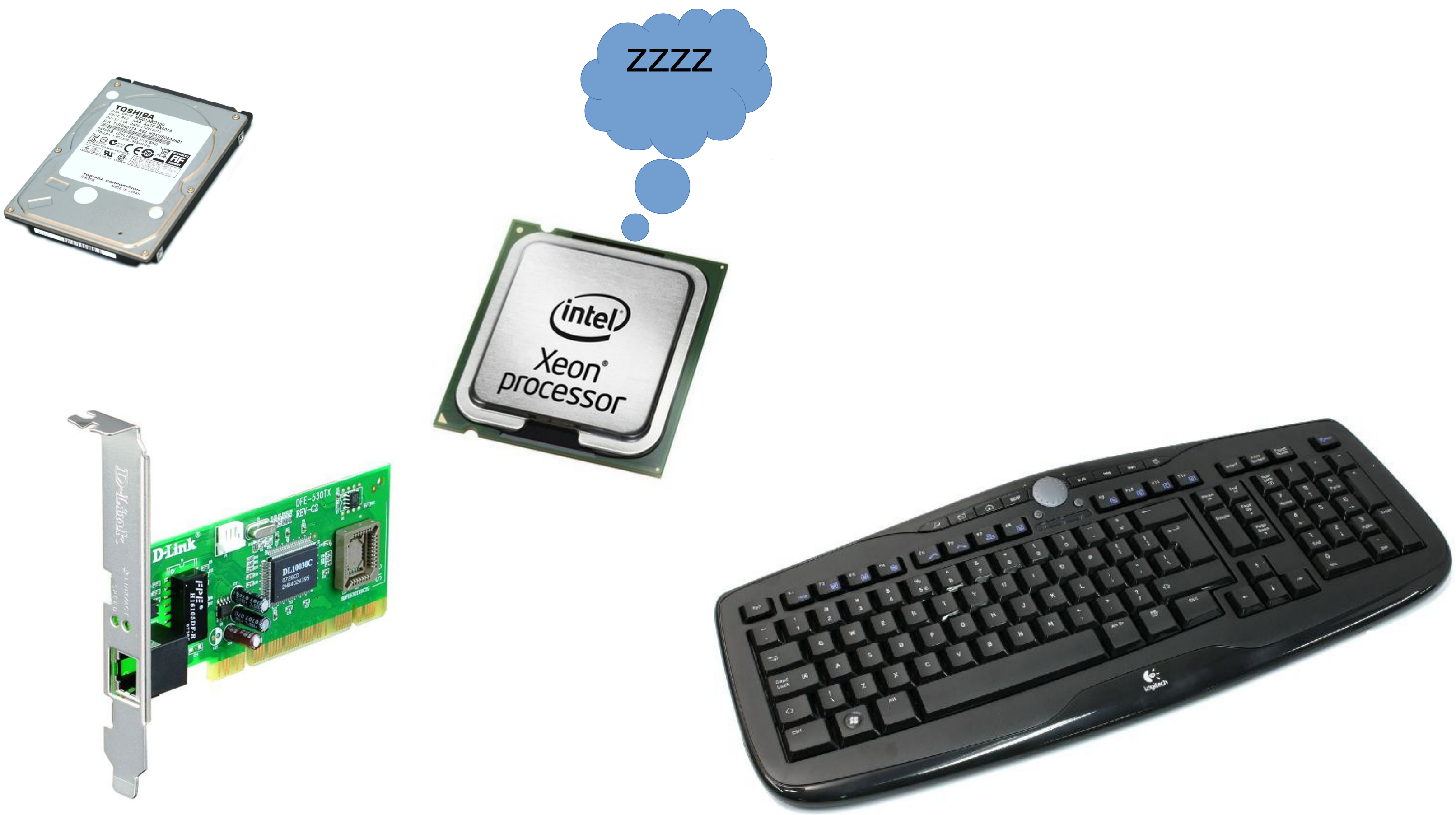
Interrupts

Interrupts: Processor wakes and fetches data (short burst)



Interrupts

Interrupts: Processor enters idle after interrupt service complete



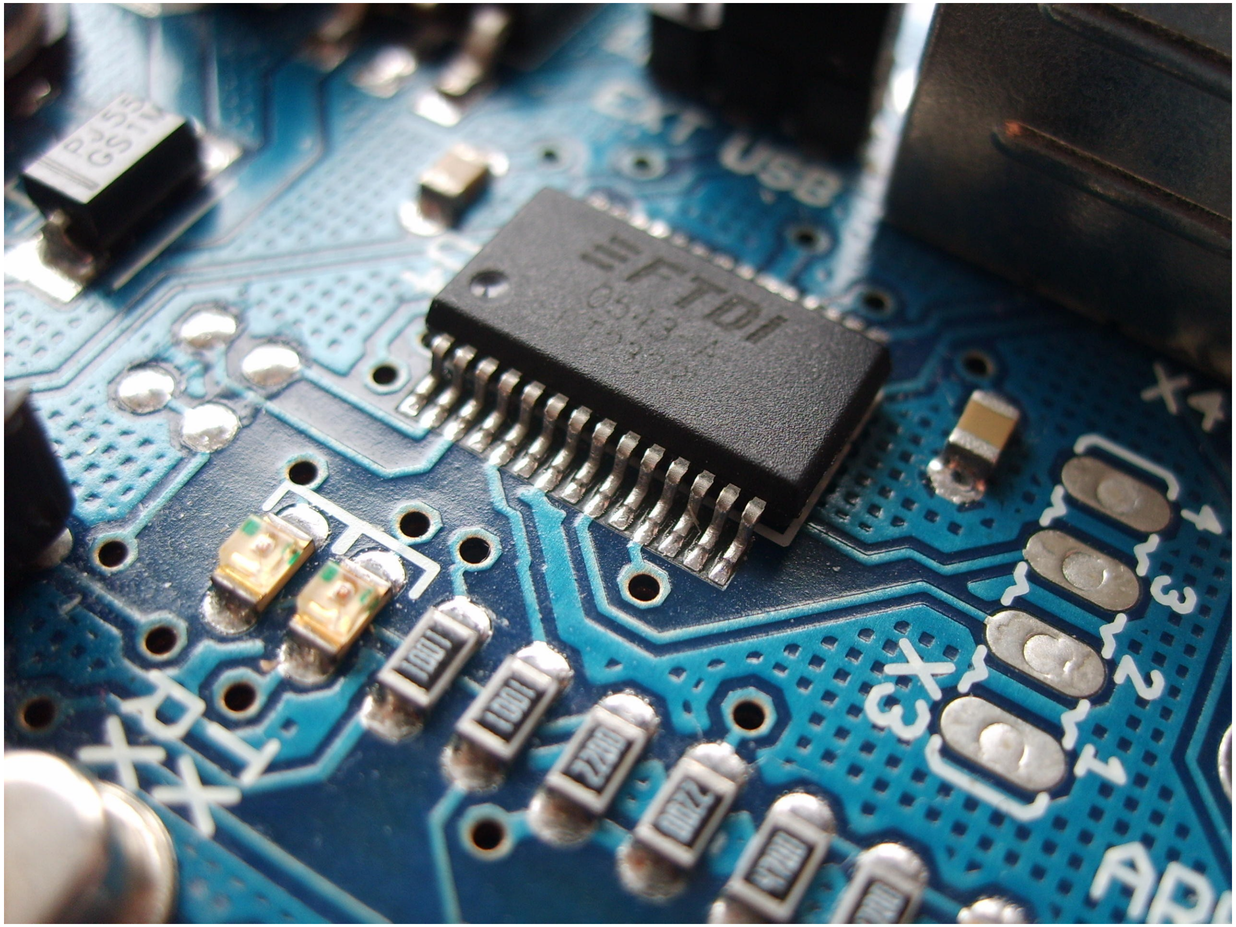
Interrupts

- Interrupts are more efficient than polling
- Reduces risk of data loss
- More complex to implement
- Allows processor execute tasks "simultaneously"

Interrupts

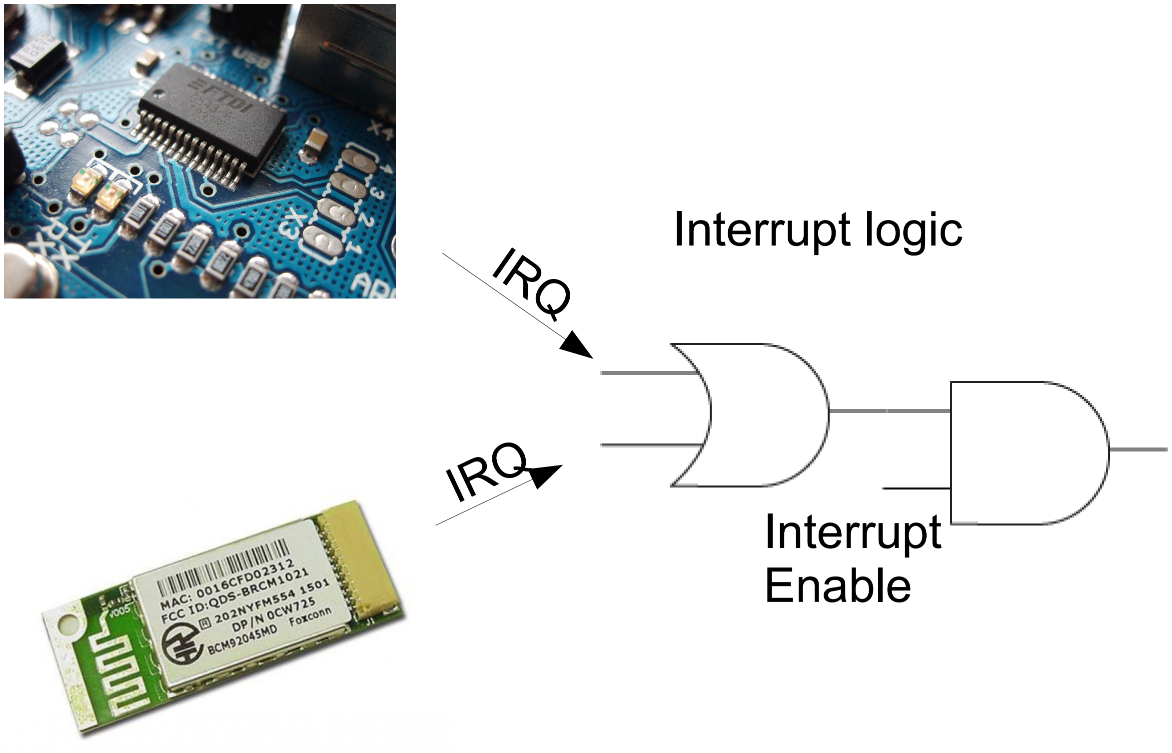
Hardware event

Interrupts



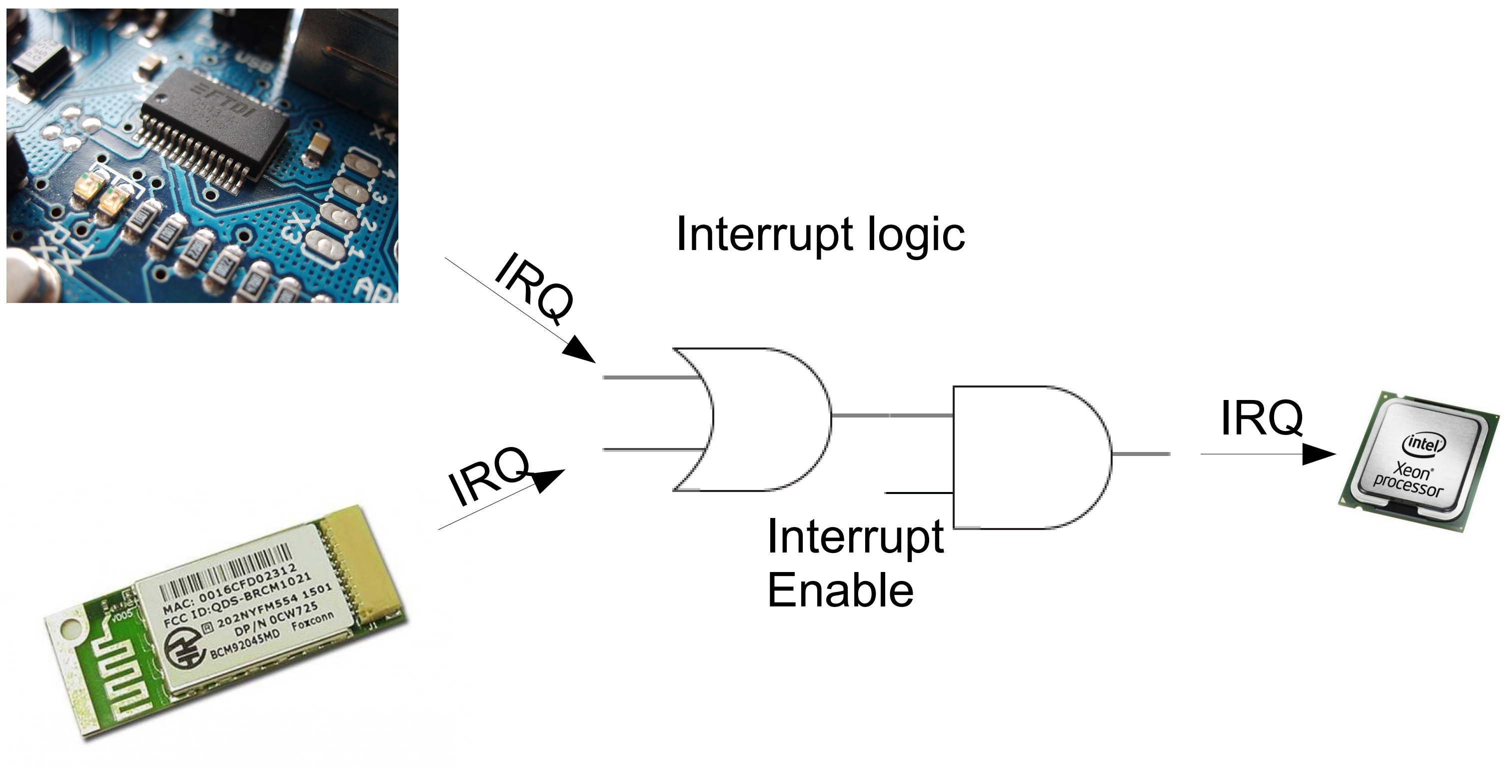
Hardware device requires attention for some reason.
Outputs a logic signal

Interrupts



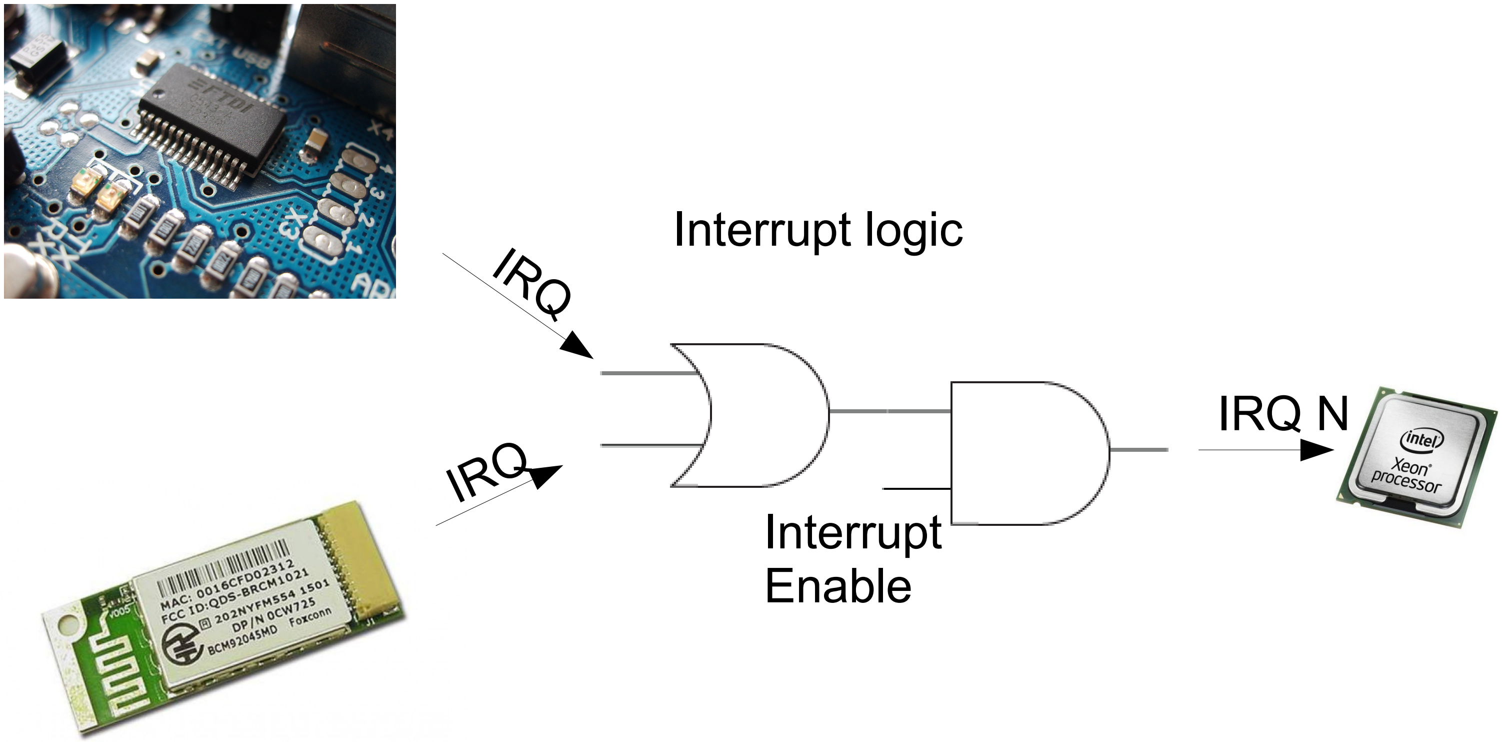
Interrupt requests are routed via interrupt logic gates

Interrupts



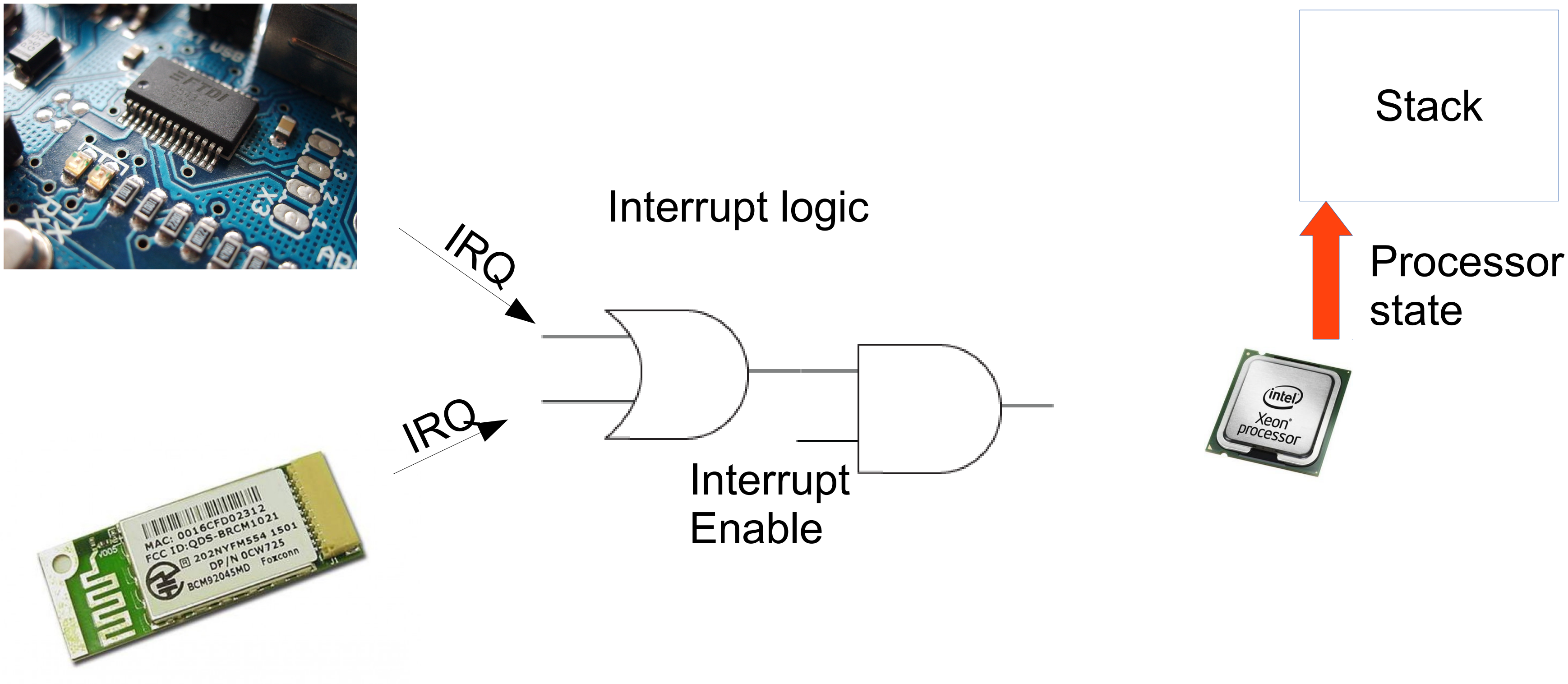
If interrupts are enabled, IRQ is passed on to CPU

Interrupts



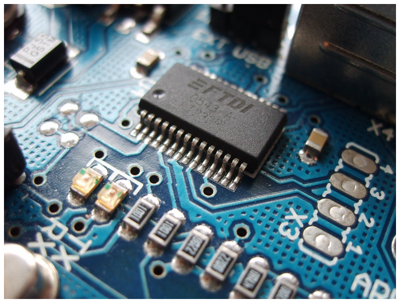
Which IRQ?
ID Number is passed to CPU (or interrupt controller)

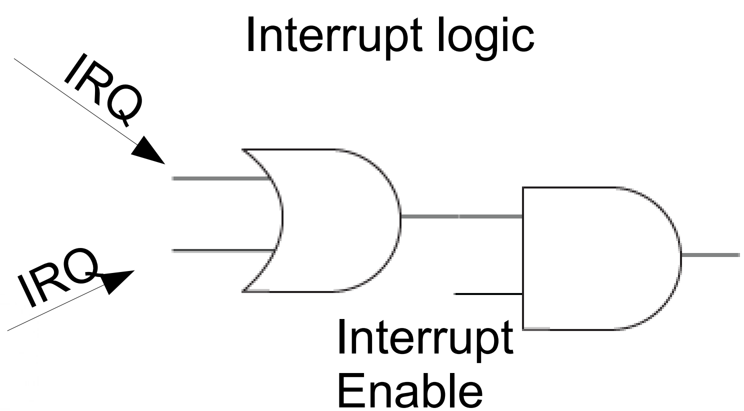
Interrupts

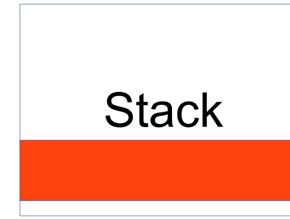


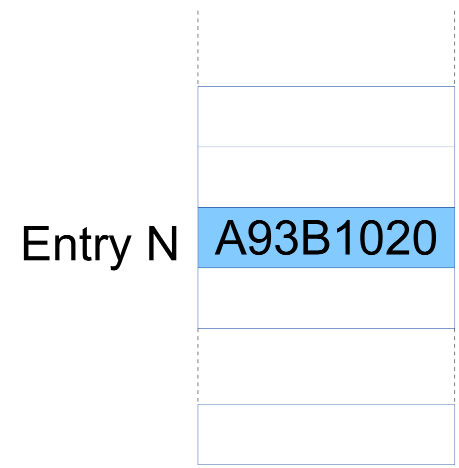
Processor stops current task and saves current state on the stack

Interrupts



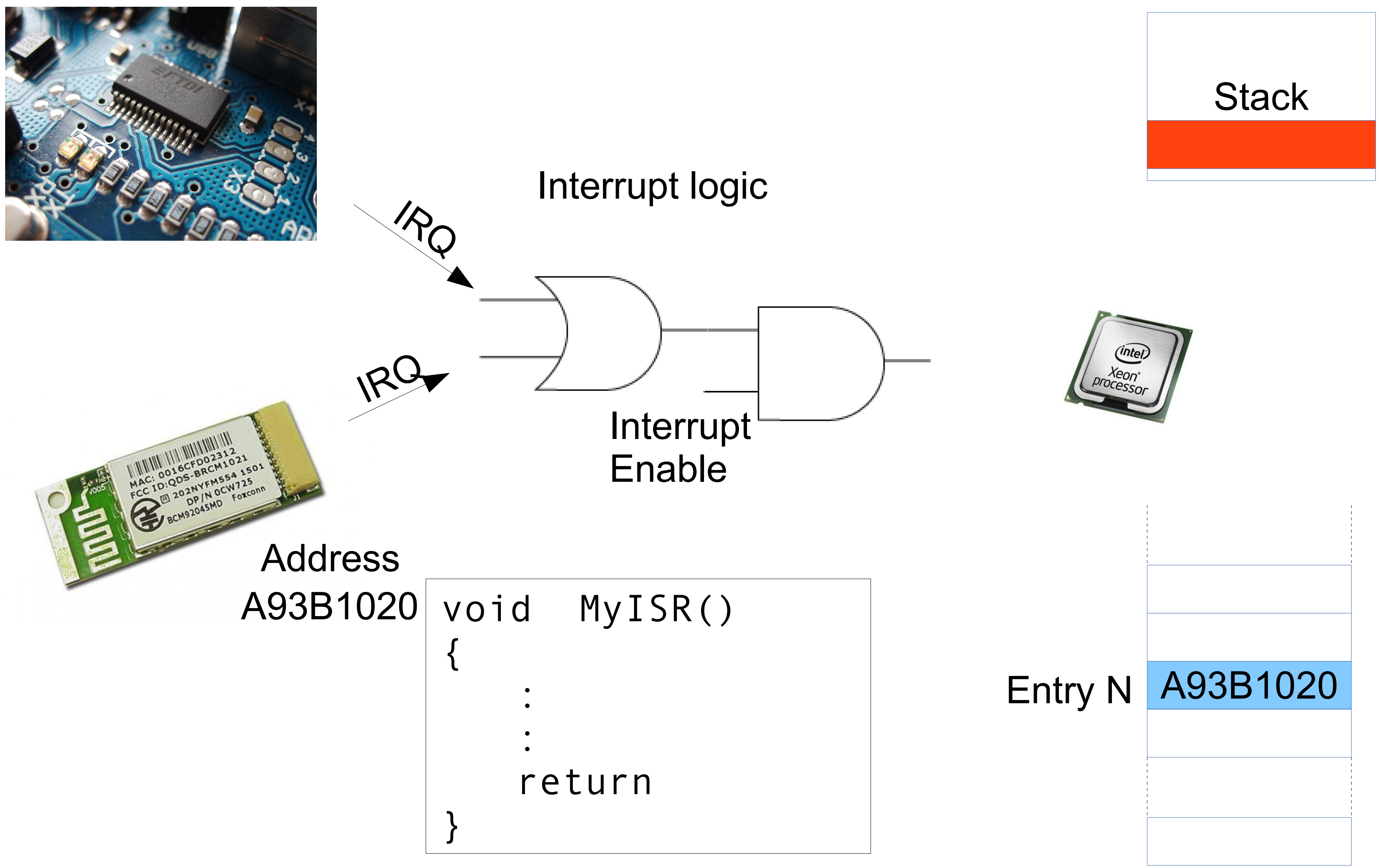






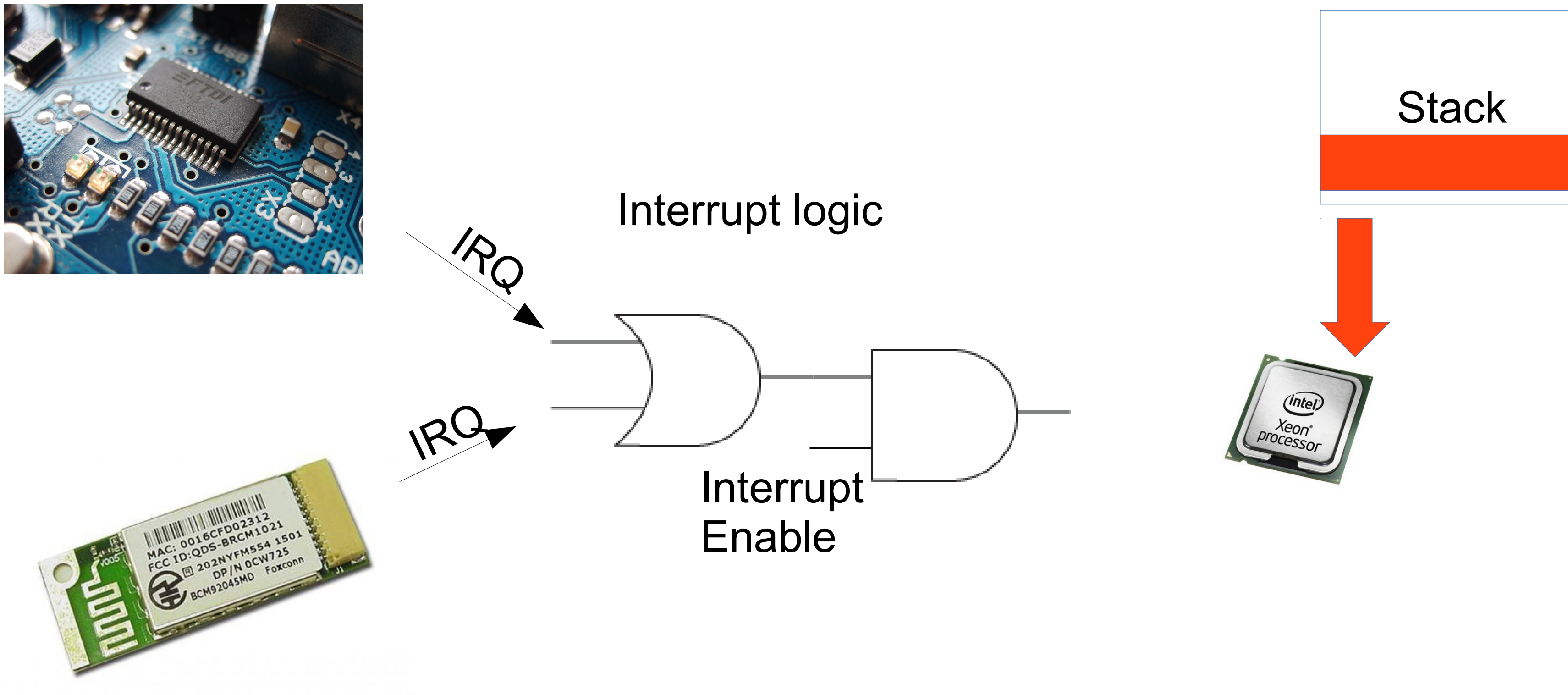
Address of interrupt handler fetched
from Interrupt Vector Table

Interrupts



Interrupt Service Routine (function) executed

Interrupts

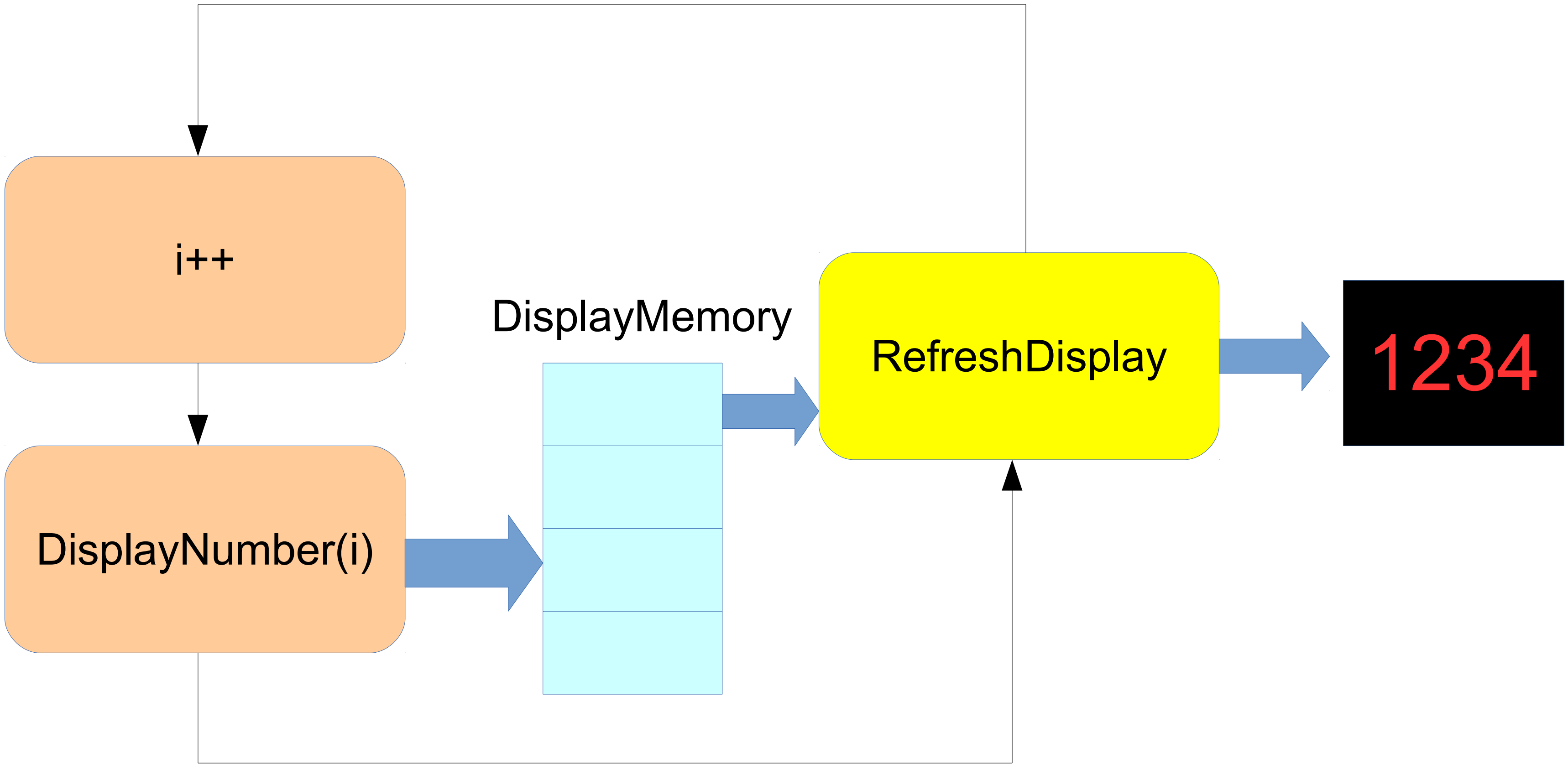


Processor state is restored
Interrupted task is resumed

Interrupts

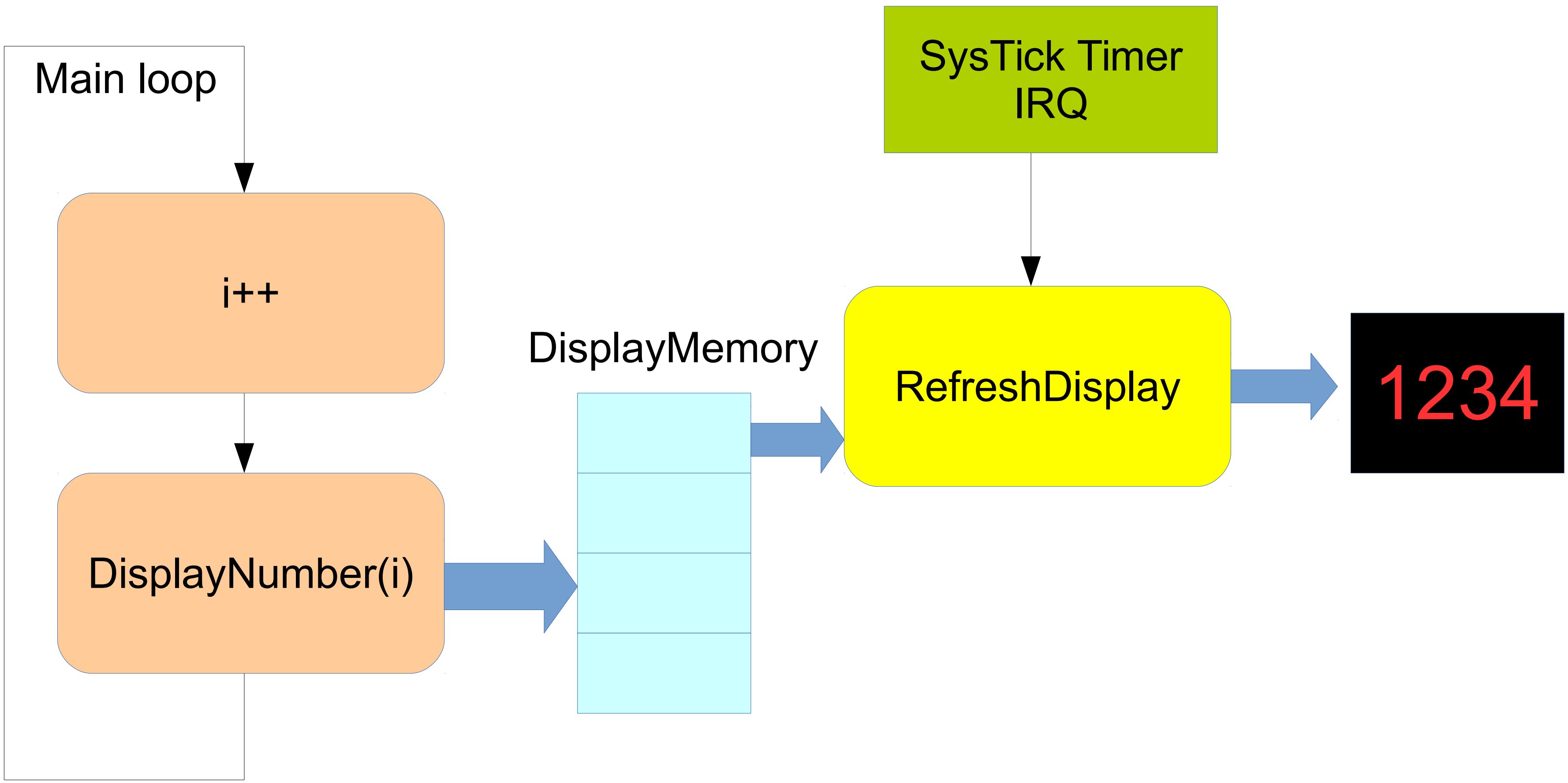
Lab 4: Using a timer interrupt to refresh a display
Lab 3 used polling (periodic calling) method to refresh display

Main loop



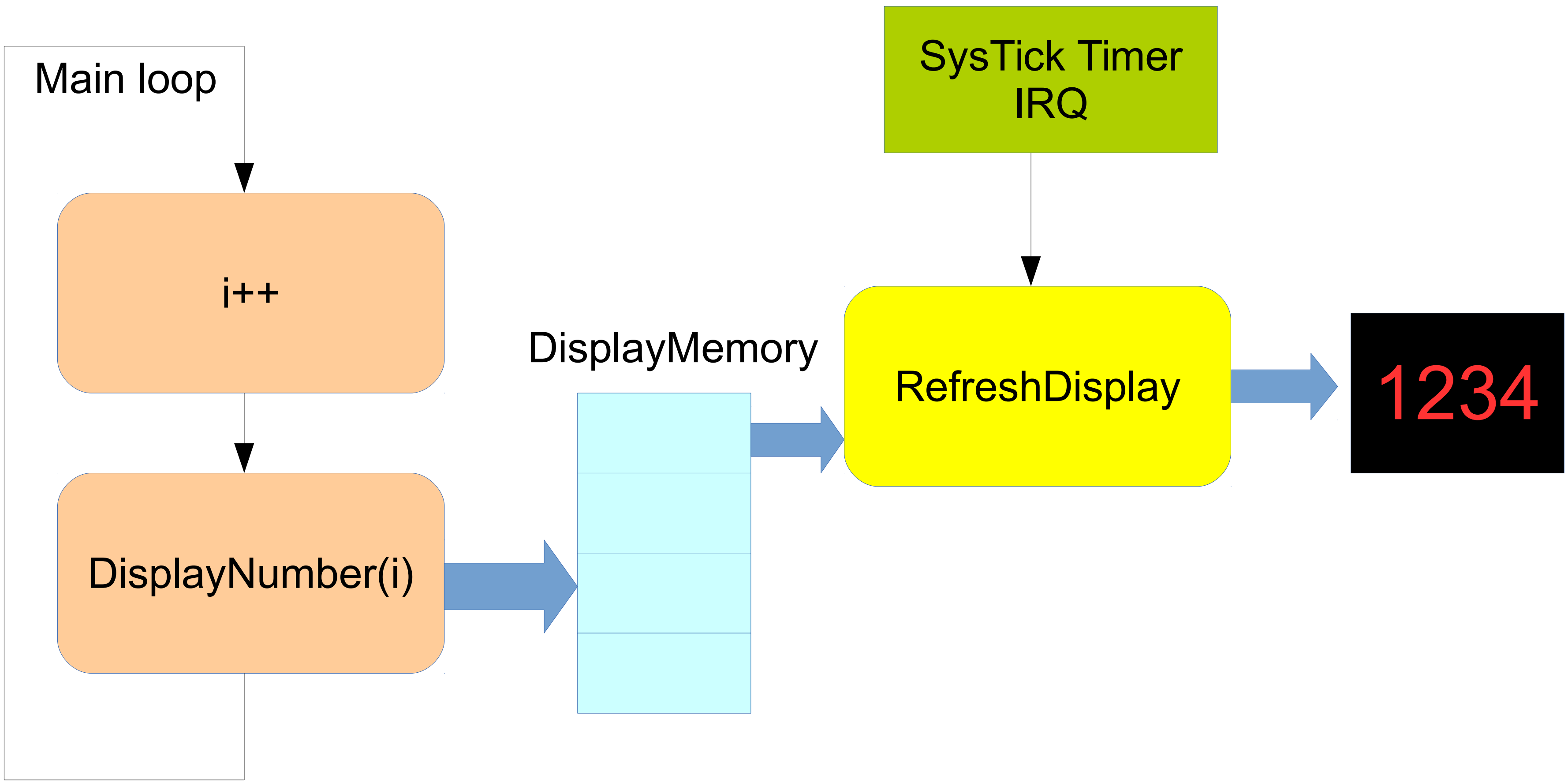
Interrupts

Lab 4: Using a timer interrupt to refresh a display
Systick timer will periodically generate an IRQ. ISR will refresh display

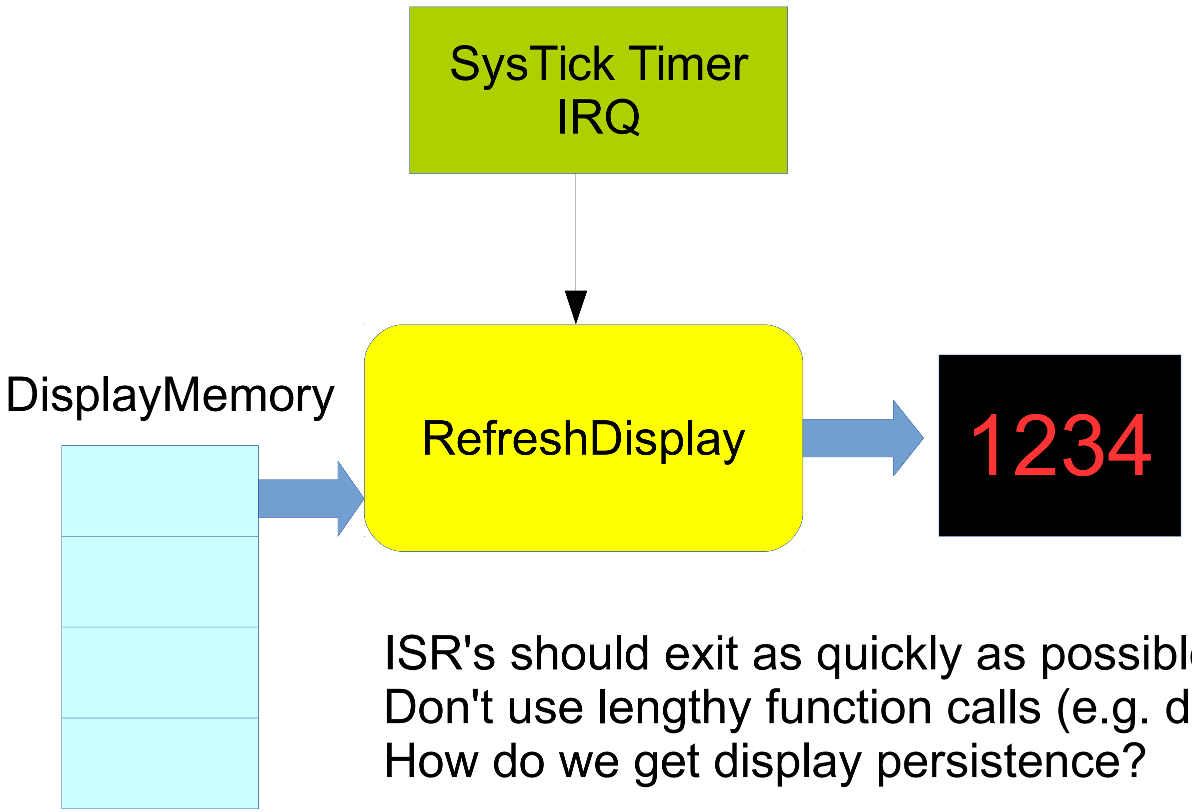


Interrupts

Lab 4: Display memory is a shared memory object. Shared by SysTick ISR and foreground program

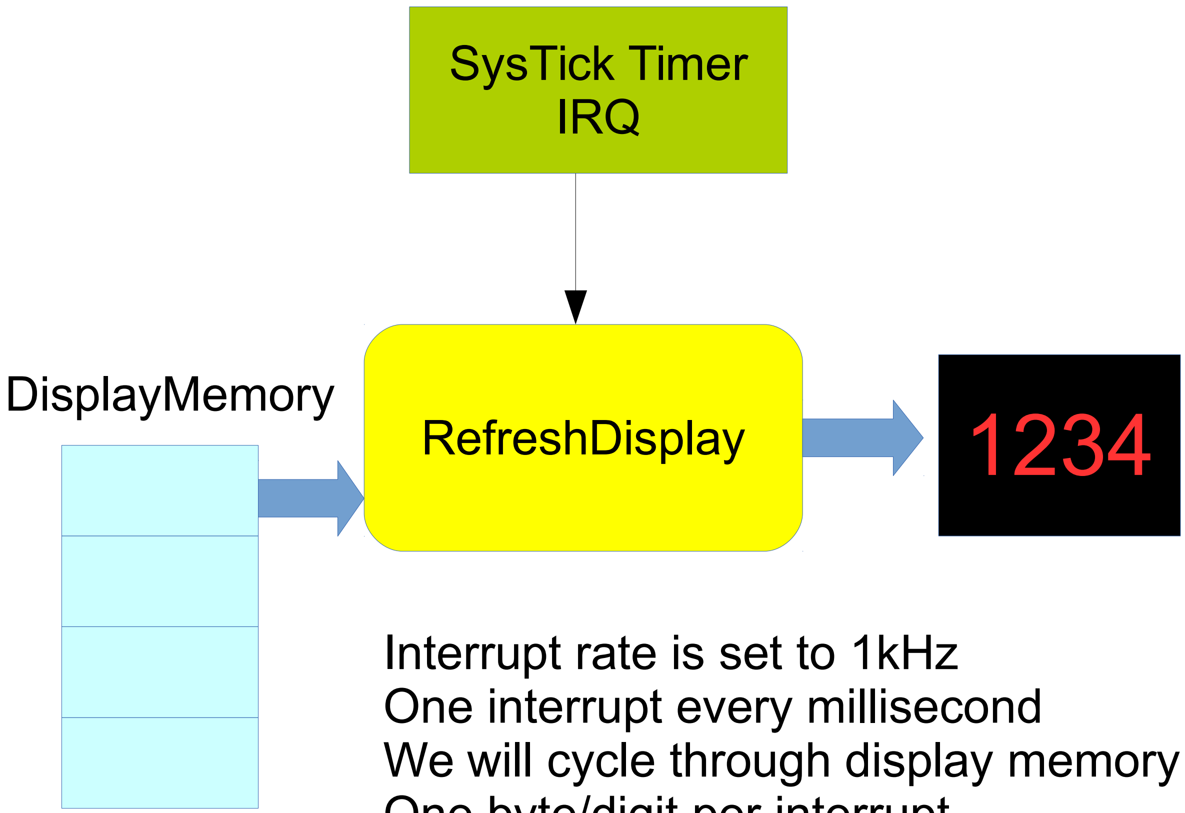


Interrupts



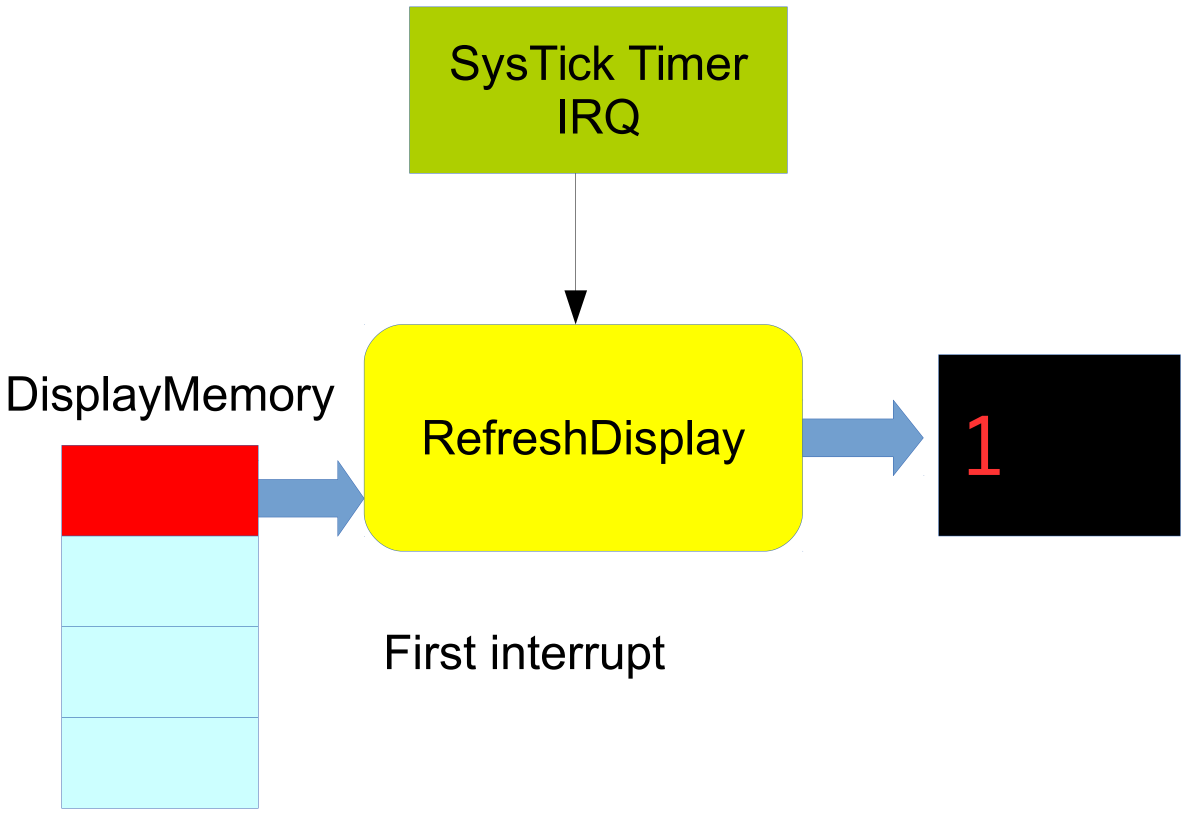
ISR's should exit as quickly as possible
Don't use lengthy function calls (e.g. delay)
How do we get display persistence?

Interrupts



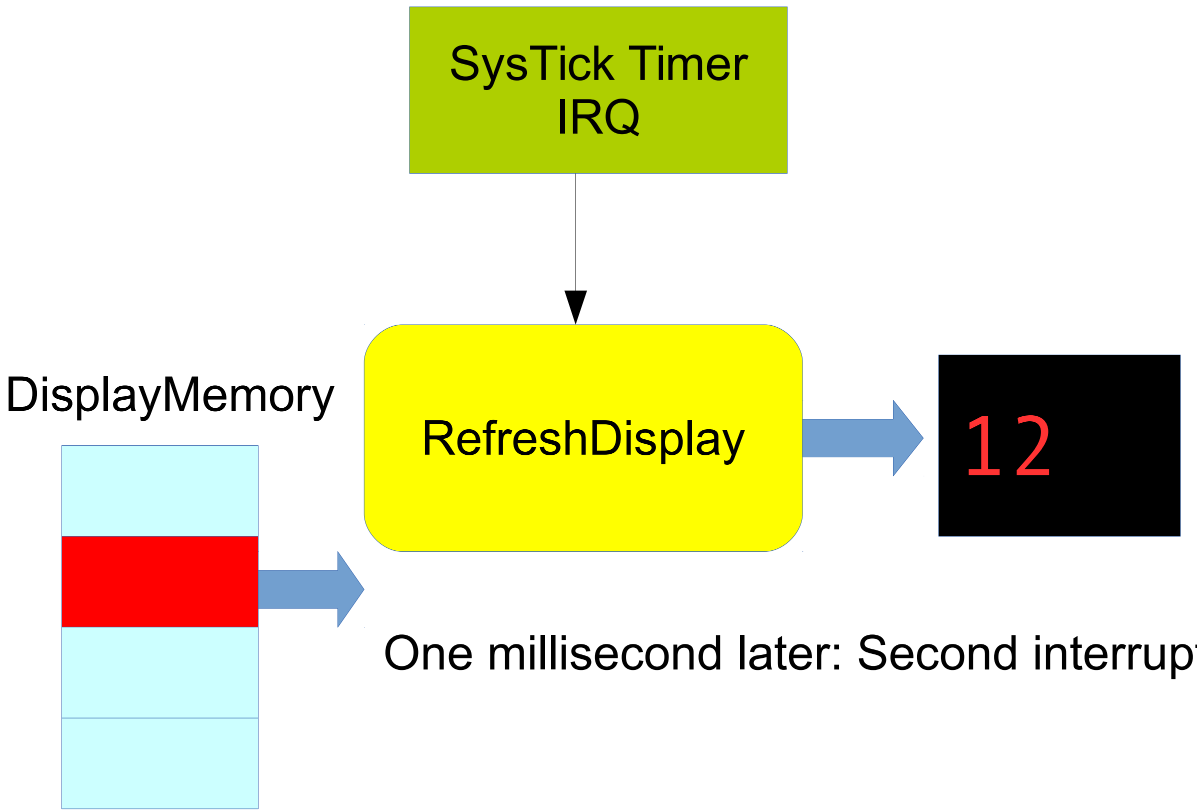
Interrupt rate is set to 1kHz
One interrupt every millisecond
We will cycle through display memory
One byte/digit per interrupt

Interrupts



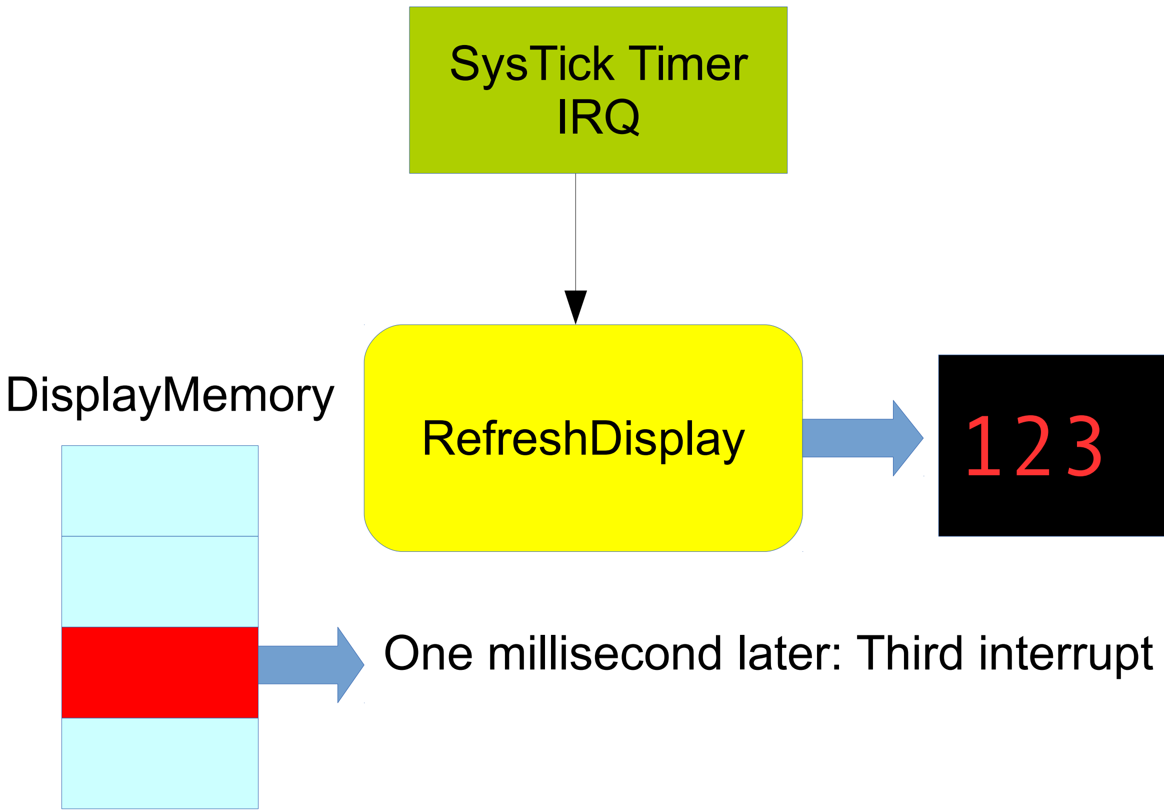
SysTick Timer
IRQ
DisplayMemory
RefreshDisplay
1
First interrupt

Interrupts



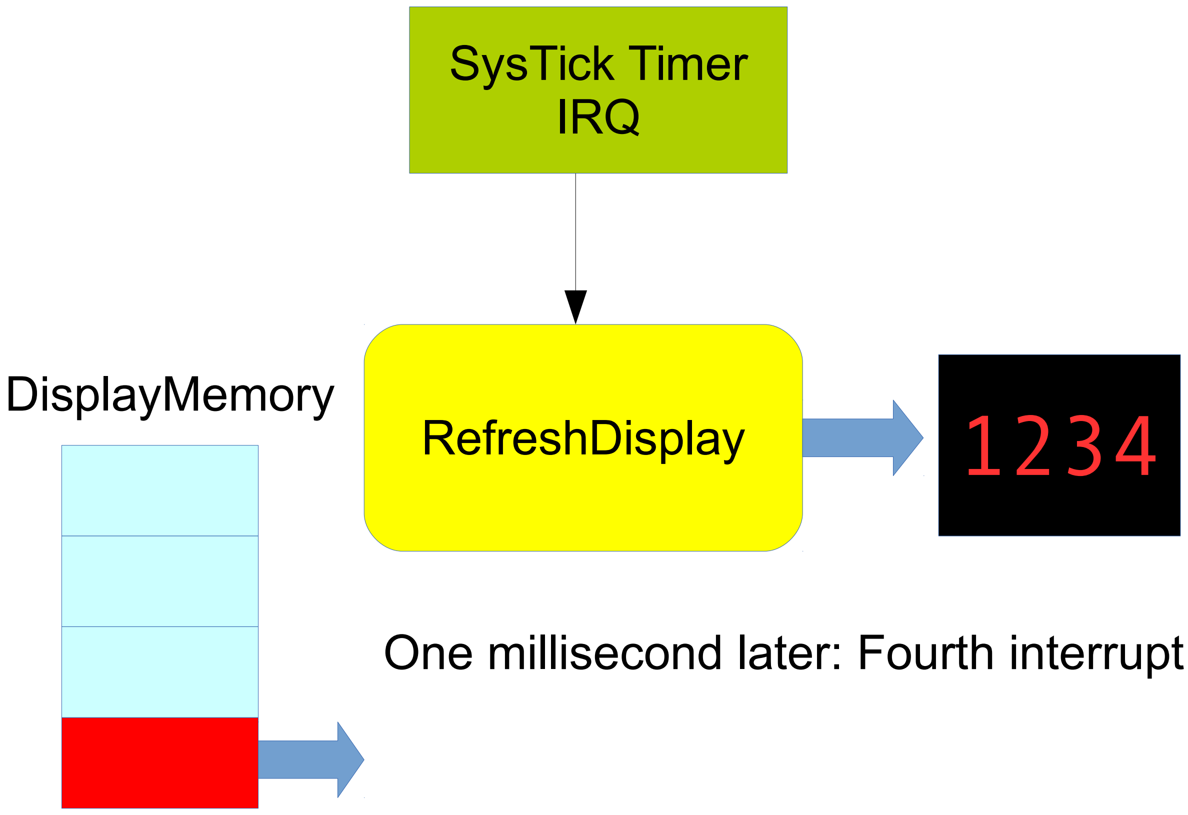
One millisecond later: Second interrupt

Interrupts



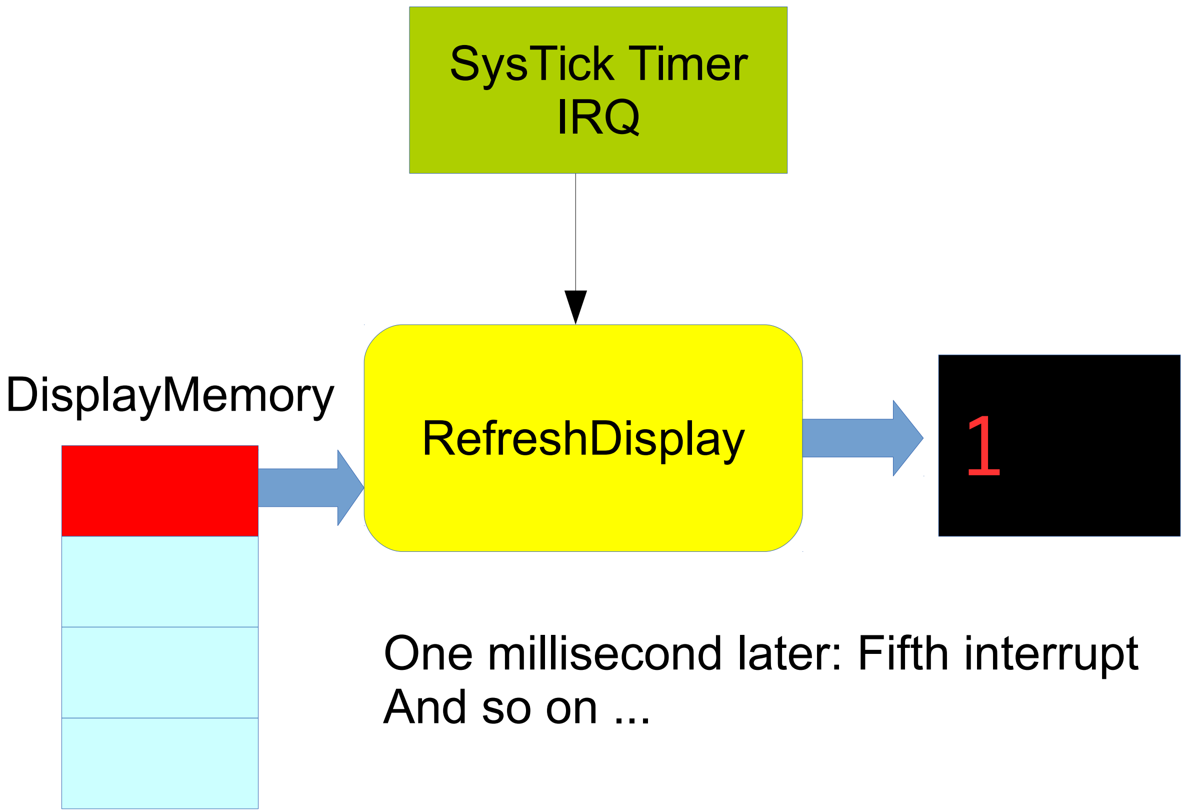
SysTick Timer
IRQ
DisplayMemory
RefreshDisplay
123
One millisecond later: Third interrupt

Interrupts



One millisecond later: Fourth interrupt

Interrupts



One millisecond later: Fifth interrupt
And so on ...

Interrupts

```
void SysTick(void)
{
// This function is triggered every millisecond by the SysTick interrupt
    static int DigitNumber=1;
    milliseconds++;
    if (milliseconds>=1000)
    {
        // A second has passed to reset the millisecond counter
        milliseconds = 0;
    }
    switch (DigitNumber)
    {
        case 1: {
            // Turn on (make low) the desired digit and blank all segments
            GPIO0DATA = DIG_1 | DIG_2 | DIG_3;
            // Set the relevant segment bits
            GPIO0DATA |= DisplayMemory[0];
            // Wait for display to light up
            break;
        }
```

Interrupts

```
case 2: {
   // Turn on (make low) the desired digit and blank all segments
   GPIO0DATA = DIG_1 | DIG_2 | DIG_4;
   // Set the relevant segment bits
   GPIO0DATA |= DisplayMemory[1];
   // Wait for display to light up
   break;
}
case 3: {
   // Turn on (make low) the desired digit and blank all segments
   GPIO0DATA = DIG_1 | DIG_3 | DIG_4;
   // Set the relevant segment bits
   GPIO0DATA |= DisplayMemory[2];
   // Wait for display to light up
   break;
```

Interrupts

```
        case 4: {
            // Turn on (make low) the desired digit and blank all segments
            GPIO0DATA = DIG_2 | DIG_3 | DIG_4;
            // Set the relevant segment bits
            GPIO0DATA |= DisplayMemory[3];
            // Wait for display to light up
            break;
        }
    }
    DigitNumber++;
    if (DigitNumber > 4)
        DigitNumber = 1;
}
```

Interrupts

```
void initSysTick()
{

    // The systick timer is driven by a 48MHz clock
    // Divide this down to achieve a 1ms timebase
    // Divisor = 48MHz/1000Hz
    // Reload value = 48000-1
    // enable systick and its interrupts
    SYST_CSR |=(BIT0+BIT1+BIT2);
    SYST_RVR=48000-1; // generate 1 millisecond time base
    SYST_CVR=5;
    enable_interrupts();
}
```

Interrupts

```
void DisplayNumber(int Number)
{
    DisplayMemory[0]=digits[Number % 10];
    Number = Number / 10;
    DisplayMemory[1]=digits[Number % 10];
    Number = Number / 10;
    DisplayMemory[2]=digits[Number % 10];
    Number = Number / 10;
    DisplayMemory[3]=digits[Number % 10];
}
```

Interrupts

```
void ConfigPins()
{
   SYSAHBCLKCTRL |= BIT6 + BIT16; // Turn on clock for GPIO and IOCON
   // Make all of the segment and digit bits outputs
   GPIO0DIR = SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | \
              SEG_G | DIG_1 | DIG_2 | DIG_3 | DIG_4;
   // Turn off (make high) all display digits
   GPIO0DATA = DIG_1 | DIG_2 | DIG_3  | DIG_4;
   // Make Port 0 bit 5 behave as a generic output port (open drain)
   IOCON_PIO0_5 |= BIT8;
   // Make Port 0 bit 10 behave as a generic I/O port
   IOCON_SWCLK_PIO0_10  = 1;
   // Make Port 0 bit 11 behave as a generic I/O port
   IOCON_R_PIO0_11  = 1;

}
```

Interrupts

```
int main()
{
    initSysTick();
    ConfigPins();
    int i=0;
    while(1)
    {
        DisplayNumber(i++);
        if (i > 9999)
            i=0;

    }
}
```

NOTE: No call to RefreshDisplay!

Interrupts

How would you extend this program to implement a time of day clock?