# Interrupts

# Functions first!

- Break code up into chunks called functions
- Why?
  - Makes code easier to organise and understand
  - If the same code is used lots of places in your program you only have to write it once and “call” it when required.

Functions

"main" program

```
int main(void)
{
    //Call function that setups up the pins
    setupPins();

    while(1)
    {
        //Call function that does something!
        doStuff();

    }

}
```

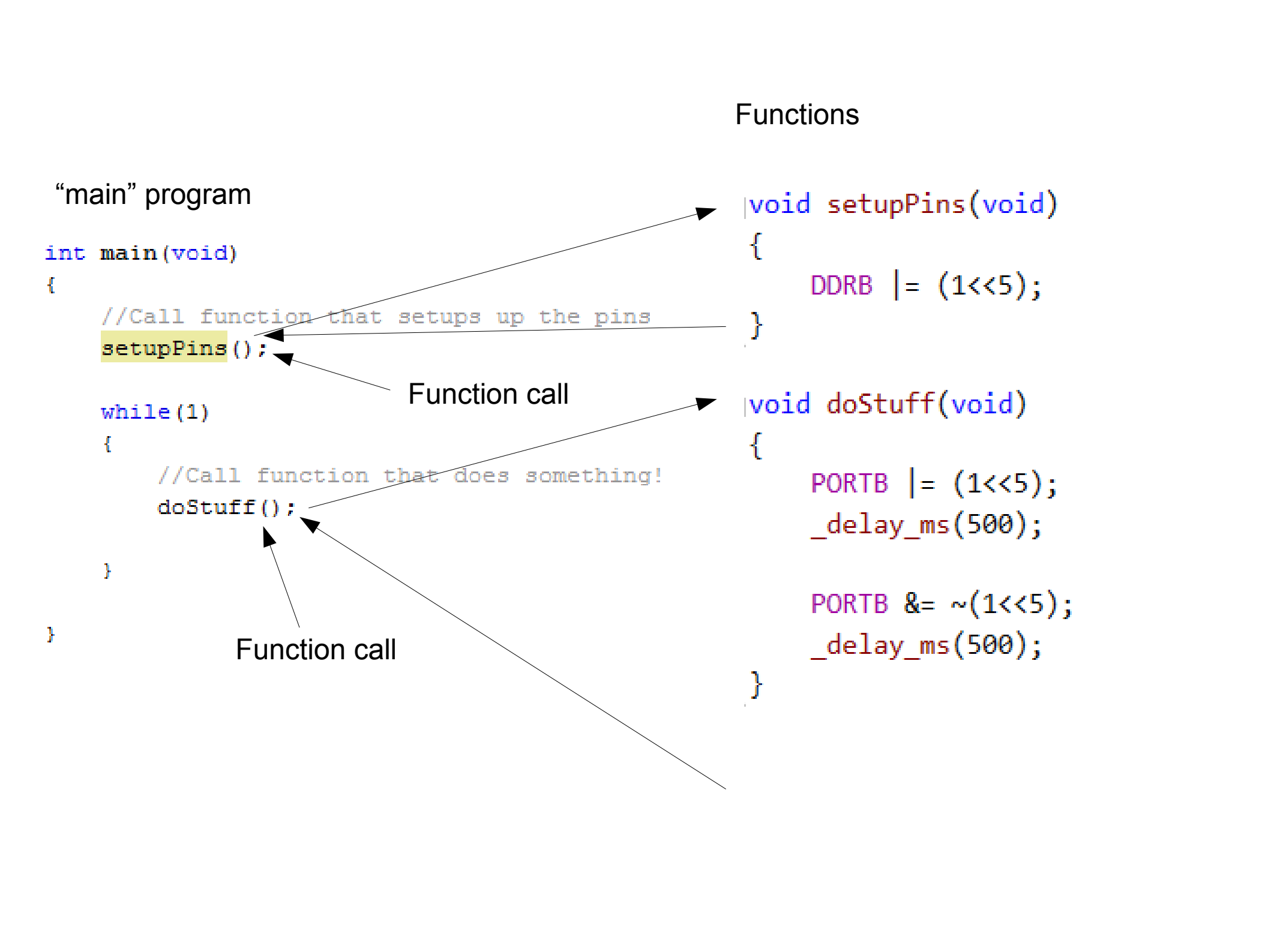
```
void setupPins(void)
{
    DDRB |= (1<<5);
}
```

```
void doStuff(void)
{
    PORTB |= (1<<5);
    _delay_ms(500);

    PORTB &= ~(1<<5);
    _delay_ms(500);
}
```

Function call

Function call



# Key points of functions

- YOU as the programmer control exactly when in your program the function executes
- How?
  - By placing the function calls where you want.

# Basic Idea of Interrupts

- Make a specific piece of code (function) run when some external hardware event occurs:
  - Button pressed
  - Timer expired
  - IR beam broken
  - Critical temperature reached
  - Break pedal pressed!

```c
int main(void)
{
    //PB5 as output pin
    DDRB |= (1<<5);

    //interrupt on rising edge
    EICRA |= (1 << ISC00);
    EICRA |= (1 << ISC01);

    // set up interrupt
    EIMSK |= (1 << INT0);   //interrupt pin enable

    //enable global interrupt -
    //set I bit in status register
    sei();

    while(1)
    {
        //doStuff
    }
}
```

Normal “main” program
Note there is NO function call

Special function that will ***only*** run when the button is pressed


```c
ISR(INT0_vect)
{
    //toggle PB5
    PORTB ^= (1<<5) ;
}
```

# Terminology

- The special function that runs when the hardware event occurs is called an ISR – Interrupt Service Routine

- The actual signal that triggers the running of the ISR is called an IRQ – Interrupt ReQuest
  - It is usually a falling or rising edge

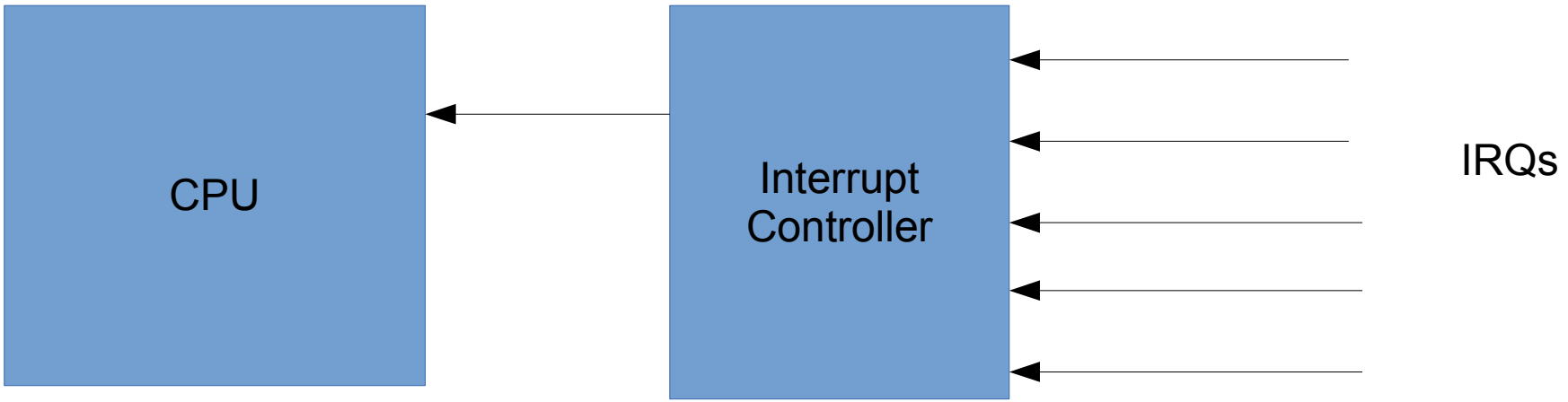
# Key points!

- There is NO function call in the main code that triggers the running of the ISR.
- What calls the ISR?
  - A hardware event – an IRQ
- The programmer has no control over when the ISR runs

# How to set up / use interrupts?

- The interrupt mechanism is built into the hardware of the CPU/MCU.
- The interrupt mechanism is disabled by default
- We have to enable the mechanism and configure it to use it
- Controlled by a set of SFR's

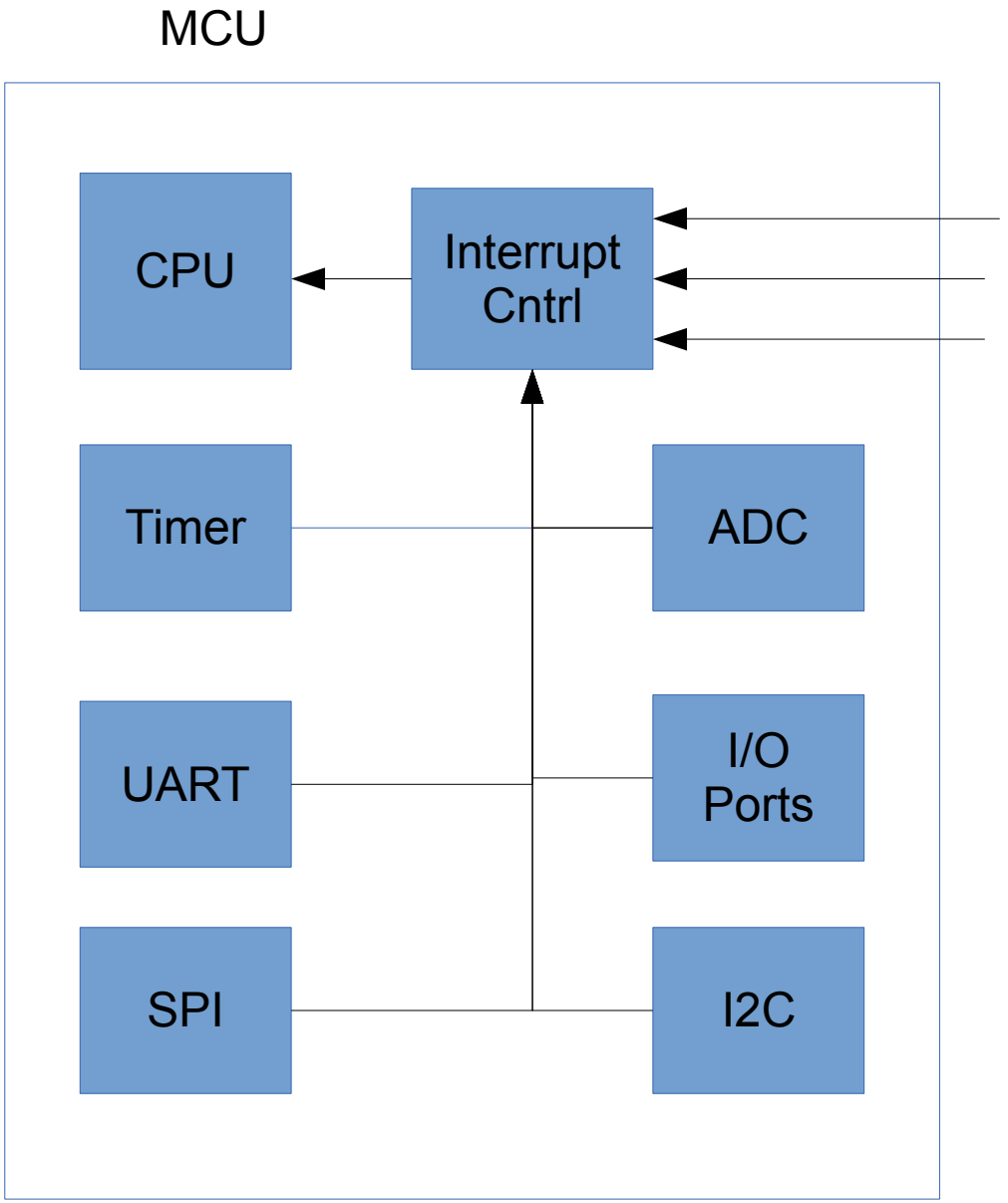
# Interrupt Mechanism




Many devices can generate IRQs

The Interrupt Controller “multiplexes” them into the CPU which only has one IRQ “pin”/input.

# Interrupt Sources




Multiple devices can Generate IRQs

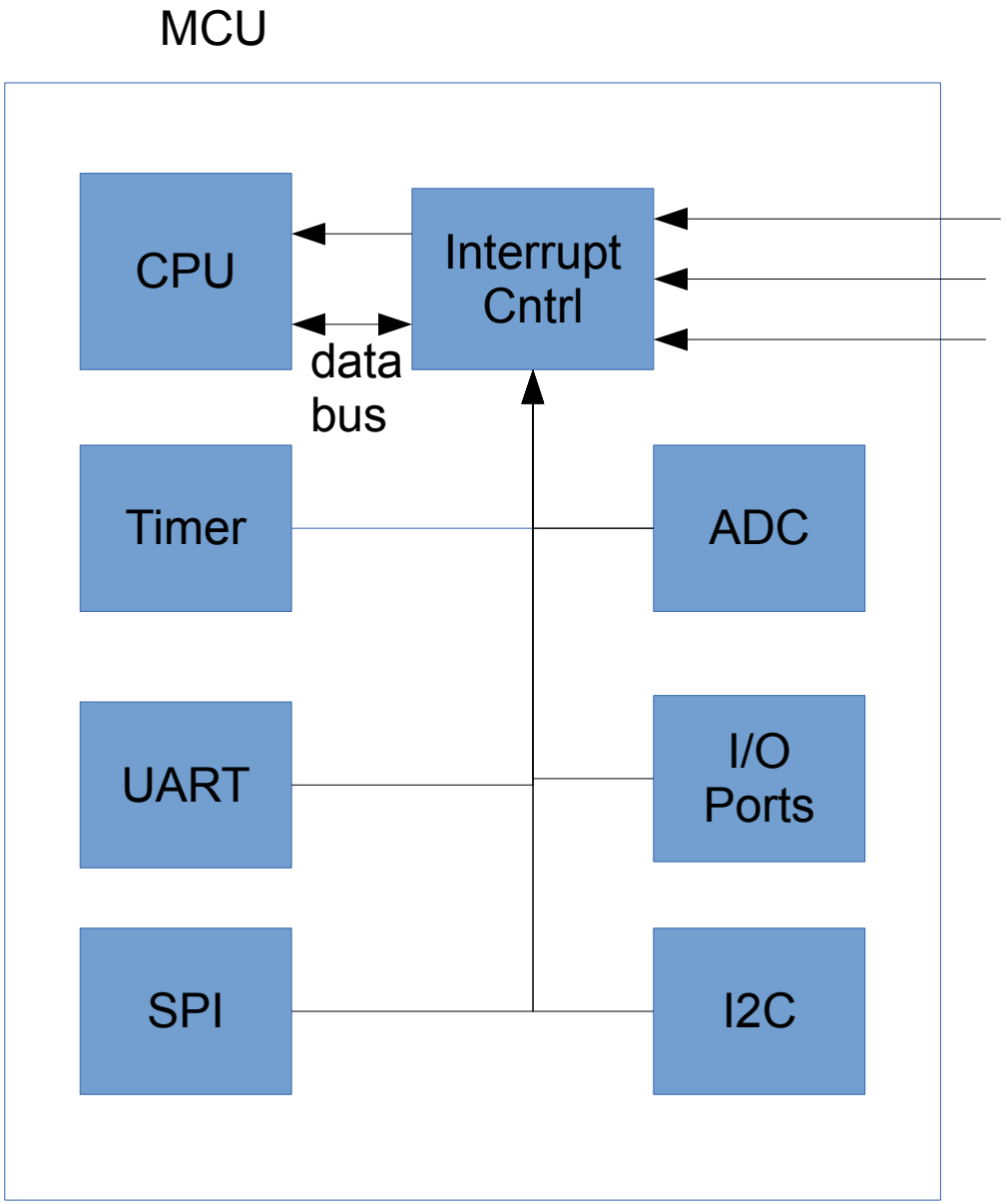
Some are internal to the MCU

Some can be external - connected by “external interrupt pins”

The interrupt controller “multiplexes” the IRQs into the CPU which has only one interrupt “pin”/input.

The interrupt controller controls which interrupts are enabled and disabled

# Interrupt Vectors




The Interrupt Controller “tells” The CPU which interrupt has occurred by passing a number to the CPU as well as the IRQ signal

This number is called an interrupt vector

SFRs inside the Interrupt Controller are used to individually enable and disable the various interrupt sources.

# Interrupt Vector table

- The ISRs for each interrupt can be placed anywhere in program memory
- So how does the CPU find the ISR when an interrupt occurs
- When an interrupt occurs the CPU will look for the address of the ISR it should run in a fixed place in memory
- This place is called the Interrupt Vector Table

# Interrupt Vector Table

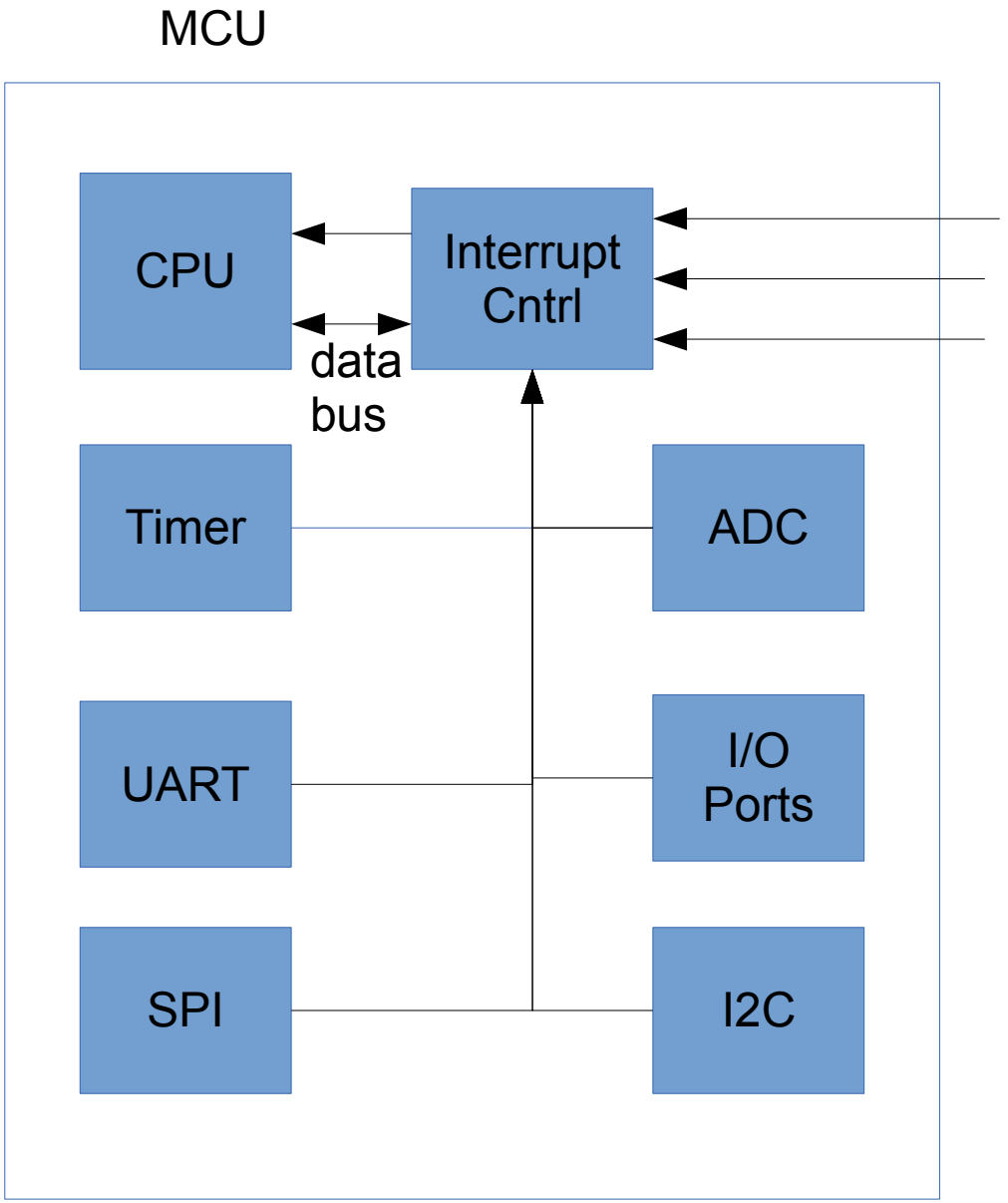
- The interrupt vector table is actually just the first chunk of program memory starting at memory address 0.
- Each separate interrupt causes the CPU to look in a particular memory address to find the address of the ISR
- When the Interrupt Controller passes an IRQ and a vector number to the CPU will use the vector number to index this table to find the address of the correct ISR
- The CPU then copies the address it finds into the Program Counter register thus forcing the CPU to jump to the code of the ISR

# Interrupt Vector Table

**Table 11-6.** Reset and Interrupt Vectors in ATmega328P

| VectorNo. | Program Address(2) | Source | Interrupt Definition |
|---|---|---|---|
| 1 | 0x0000(1) | RESET | External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset |
| 2 | 0x0002 | INT0 | External Interrupt Request 0 |
| 3 | 0x0004 | INT1 | External Interrupt Request 1 |
| 4 | 0x0006 | PCINT0 | Pin Change Interrupt Request 0 |
| 5 | 0x0008 | PCINT1 | Pin Change Interrupt Request 1 |
| 6 | 0x000A | PCINT2 | Pin Change Interrupt Request 2 |
| 7 | 0x000C | WDT | Watchdog Time-out Interrupt |
| 8 | 0x000E | TIMER2 COMPA | Timer/Counter2 Compare Match A |
| 9 | 0x0010 | TIMER2 COMPB | Timer/Counter2 Compare Match B |
| 10 | 0x0012 | TIMER2 OVF | Timer/Counter2 Overflow |
| 11 | 0x0014 | TIMER1 CAPT | Timer/Counter1 Capture Event |
| 12 | 0x0016 | TIMER1 COMPA | Timer/Counter1 Compare Match A |
| 13 | 0x0018 | TIMER1 COMPB | Timer/Coutner1 Compare Match B |
| 14 | 0x001A | TIMER1 OVF | Timer/Counter1 Overflow |
| 15 | 0x001C | TIMER0 COMPA | Timer/Counter0 Compare Match A |
| 16 | 0x001E | TIMER0 COMPB | Timer/Counter0 Compare Match B |
| 17 | 0x0020 | TIMER0 OVF | Timer/Counter0 Overflow |
| 18 | 0x0022 | SPI, STC | SPI Serial Transfer Complete |

# Interrupt Priorities




There are 26 possible interrupt sources on the 328p.

If two or more interrupts arrive at the same moment in time and have the same priority levels then the Int Controller will select the one with the highest “natural priority” first

The natural priorities are built into the MCU and found in the datasheet.

The lower the vector number the higher the priority.

# Ignoring Interrupts

- Why?
  - Doing important work in the main code.

- Different architectures do it differently but all have some mechanism to allow to ignore or disable interrupts
- On the AVR we can indiviudally enable or disable each interrupt source – basically control each line into the Int Controller..
- There is also a “global” bit in the status register – the I bit.  Basically controls the line into the CPU.

# Ignoring Interrupts

- Not really ignored – just delayed.
- Flag gets set and CPU will deal with interrupts once interrupts are re-enabled.

# How to write Interrupt based Code

- We need to identify which SFR's in the Interrupt Controller we need to:
  - Enable the required interrupt source
- Write the ISR

# Getting the the ISR address into the Vector Table

- If you were working in assembly it would be your (the programmer's) responsibility to put the ISR address into the vector table – gnarly!
- Working in C is easy – the complier does it for us once it knows we are writing an ISR
  - EVERY compiler has it's own way of doing this
  - Non-standard C
  - Must learn the way for your compiler

# Interrupt Flags

- One detail that must be taken care of in the ISR is to clear the Interrupt flag
- This is the flag inside the Interrupt Controller that is set when the source IRQ comes in
  - One for each interrupt source
  - Often the responsibility of ISR programmer to clear this flag
- In the AVR this flag is cleared by hardware – no need to handle in the code.

# Summary of writing Interrupt Code

- In main code before while(1) loop configure and enable interrupts:
  - SFR's to decide on rising/falling edge trigger
  - SFRs to enable int source
  - Globally enable interrupts – sei().
- Write ISR
  - Compiler specific code

# Why Interrupts?

- Makes code easier to write – honestly!
- Makes code respond very fast
- Can do clever things like put CPU to “sleep” (low power mode) until and interrupt arrives to wake it up – saves battery
- Embedded Systems usually rely on interrupts