PATRIoT lab

# Lecture 5 – Interrupts and Timers

Dr. Aftab M. Hussain,

Assistant Professor, PATRIoT Lab, CVEST

# Need for interrupts

- Interrupts are exactly what the term means
- This is a very natural way to respond to unexpected events or random events
- We routinely respond to interruptions in our daily lives and then resume the original activity

# Need for interrupts

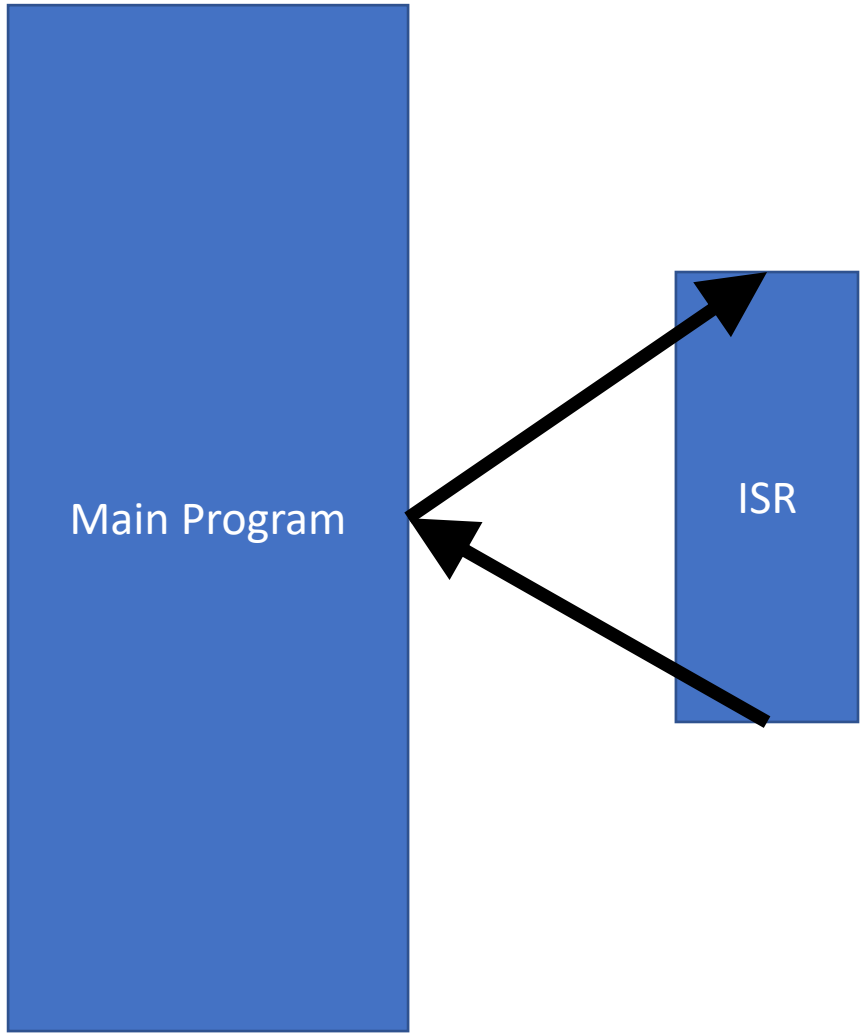
- Interrupts are often used to capture events that can occur at *random* in the external world
- The most common example is of a button being pressed
- The software for what happens when the button is pressed is known, but when to execute it is unknown – the time of the press
- Interrupts can be used to separate non-time-critical functions (within the main loop) from the time-critical functions that are executed on demand in response to interrupts

# Introduction

- An interrupt is the automatic transfer of software execution in response to an event that is asynchronous with the current software execution
- This event is called a trigger
- The event can either be a busy to ready transition in an external I/O device (like the UART input/output) or an internal event (like bus fault, memory fault, or a periodic timer)
- This should interrupt the flow of the embedded system program to follow another set of instructions
- These instructions are called the interrupt service routine (ISR)
- Once the ISR is completed, the code is expected to execute at the point of break

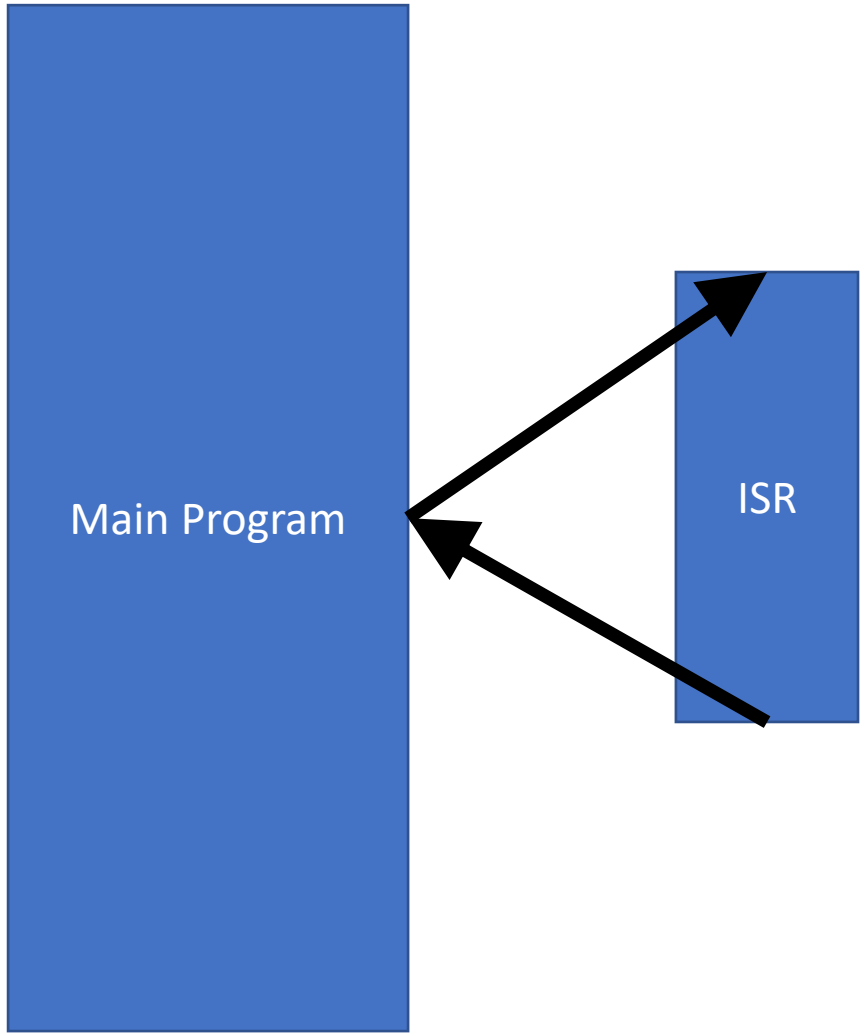
# How they work

- The ISR is stored in a specific part of the memory with the address of the ISR stored in dedicated registers
- When the interrupt is triggered, the current state of the program has to be stored temporarily and the program execution needs to move to the ISR address
- Following the completion of the ISR, the program execution returns to the original position



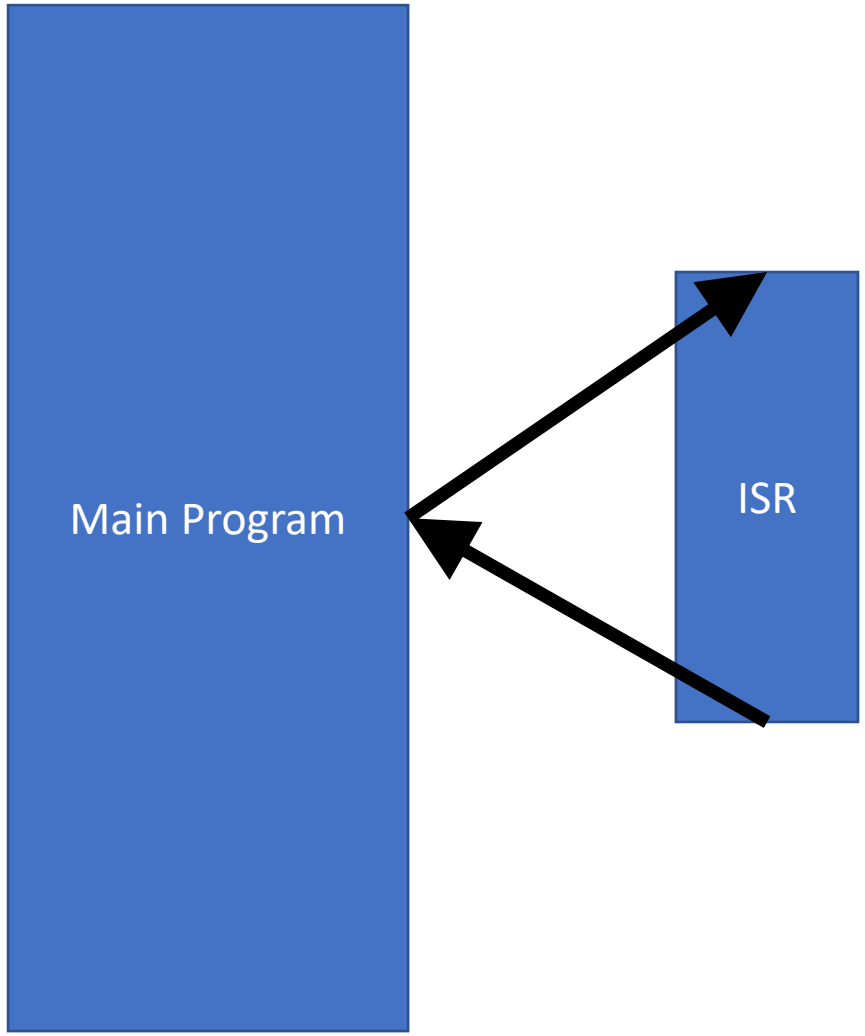
# How they work

- The controller needs to recognize that the code involves possibilities of an interrupt
- This is done by using an enabling interrupts function generally provided as standard for any controller supporting interrupts
- This makes the controller check for the trigger flag for the interrupts that are enabled and for the type of interrupts in play
- Once an interrupt is being processed, other interrupts are automatically disabled (except in special cases)

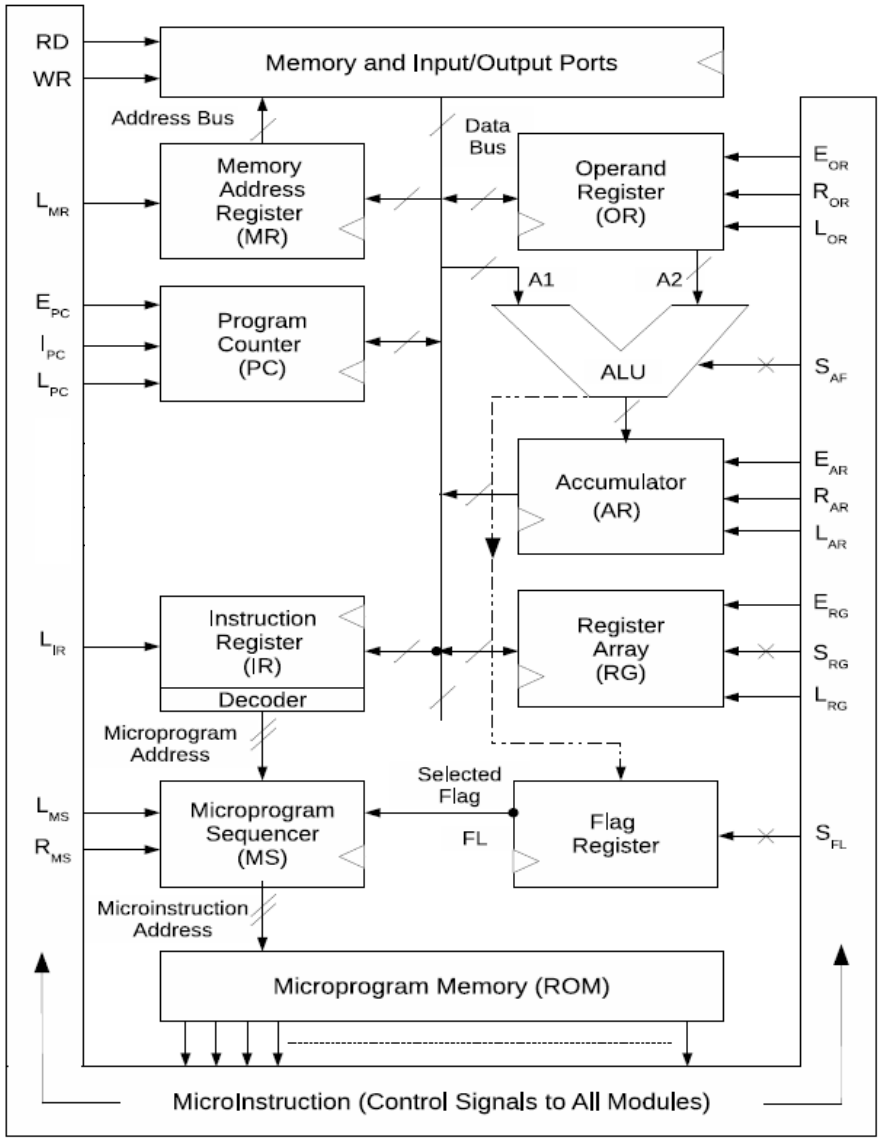


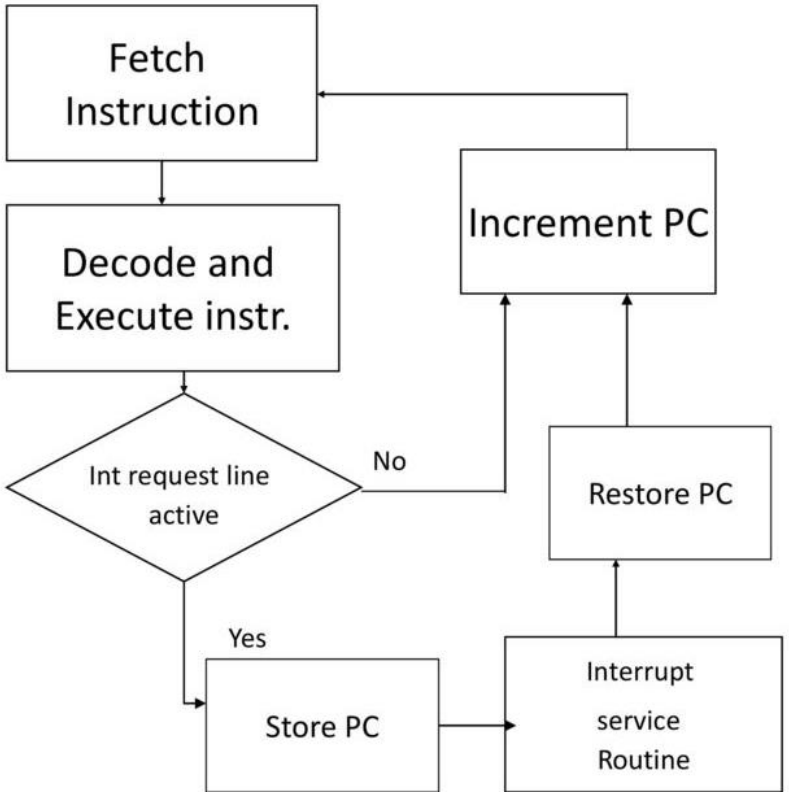
# Interrupt setup

- Apart from the global interrupt enable, many other registers have to be set
- Interrupt control register – used for configuring the interrupt pin, type of interrupt, level, edge triggered etc.
- Interrupt priority – used to determine which interrupt are serviced if more than one occur simultaneously



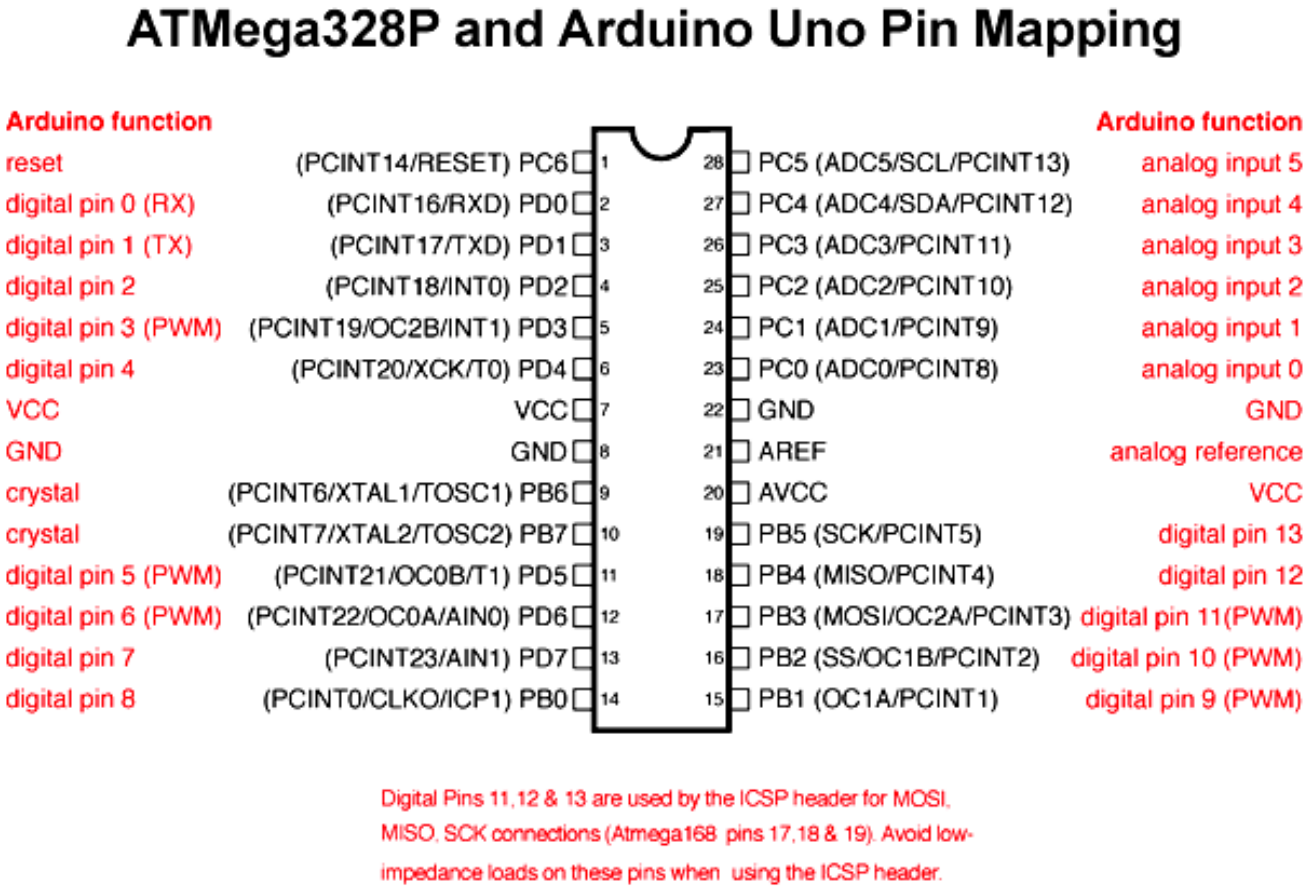
# Interrupt setup



# Interrupts in Arduino

- External interrupts
  - Used to monitor changes in I/O pins
  - The triggering event can be configured depending on the application using the control register
- On Arduino uno, there are two external interrupts INT0 and INT1 at pin 2 and 3 that can be used for multiple event types
- In atmega328, all of the IO pins can be used as pin change interrupts (PCINT) to detect toggle in signal level



# Interrupts in Arduino

- To use INT0 and INT1, the corresponding bits need to be set in the GICR
- The register is set when the controller encounters the instruction determining the use of one of the external interrupt pins
- If this bit is set along with the global interrupt enable, the processor checks for interrupts (interrupt flag) after every instruction

**General Interrupt Control Register – GICR**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Bit Name | INT1 | INT0 | – | – | – | – | IVSEL | IVCE |
| Read/Write | RW | RW | RW | RW | RW | RW | RW | RW |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Interrupts in Arduino

- Then we need to set the external interrupt control register (EICR), sometimes also referred to as MCUCR
- This determines the trigger event for the interrupts
- The interrupt is triggered only if:
  - The global interrupt bit is set
  - The corresponding GICR bit is set
  - The MCUCR is set
  - The correct event occurs at the external pin

MCU Control Register– MCUCR

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Bit Name | SE | SM2 | SM1 | SM0 | ISC11 | ISC10 | ISC01 | ISC00 |
| Read/Write | RW | RW | RW | RW | RW | RW | RW | RW |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

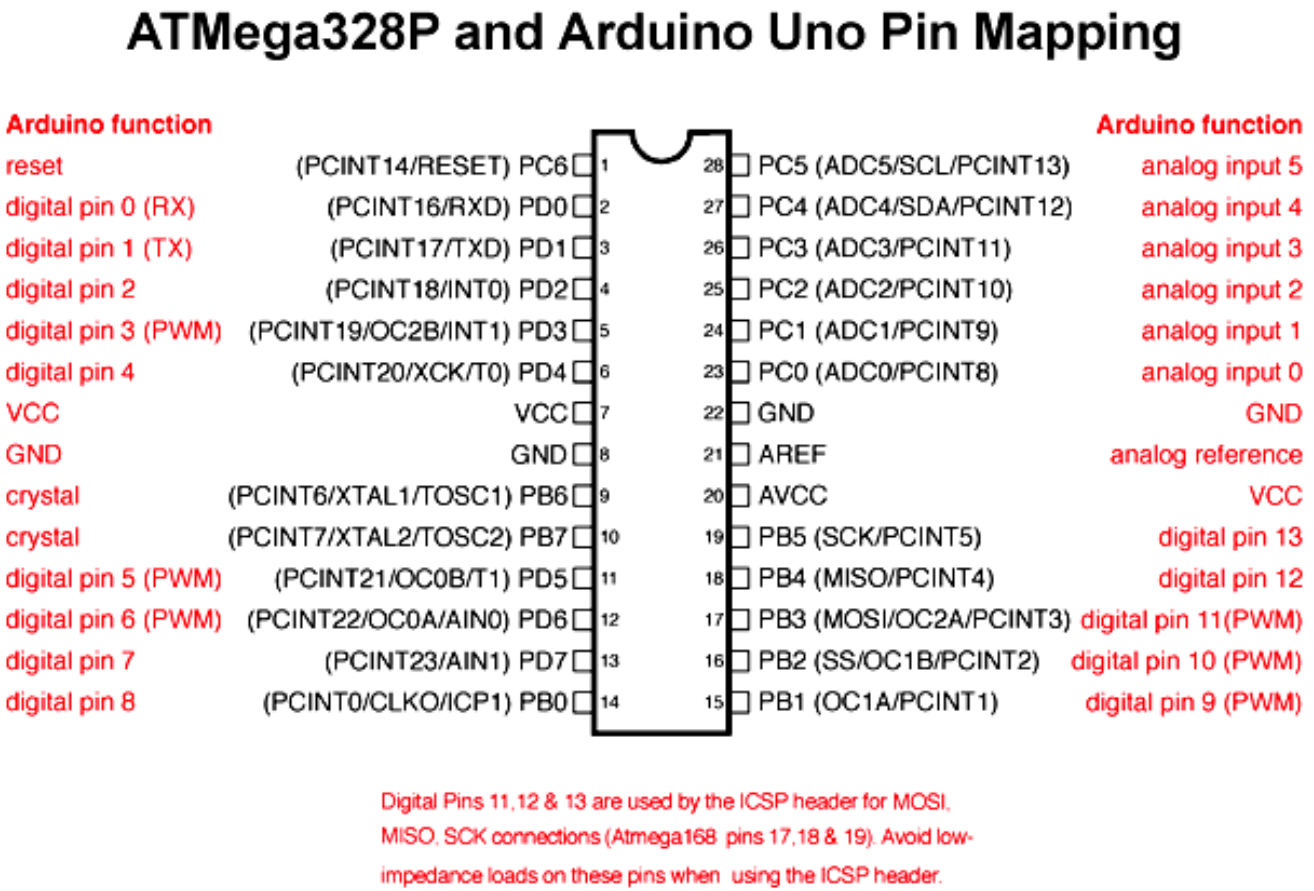
# Interrupts in Arduino

| Interrupt Sense Control | | |
|---|---|---|
| **ISCx1** | **ISCx0** | **Interrupt Generated Upon** |
| 0 | 0 | The low Level of INTx pin |
| 0 | 1 | Any logical change in INTx pin |
| 1 | 0 | Falling edge of INTx |
| 1 | 1 | Rising edge of INTx |
| x- Interrupt number, either 0 or 1 | | |

| MCU Control Register– MCUCR | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Bit** | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **Bit Name** | SE | SM2 | SM1 | SM0 | ISC11 | ISC10 | ISC01 | ISC00 |
| **Read/Write** | RW | RW | RW | RW | RW | RW | RW | RW |
| **Initial value** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Interrupts in Arduino IDE

- attachInterrupt(digitalPinToInterrupt(pin), ISR, mode)
- This single line of code in Arduino IDE does all of the required bit setting in the controller
- The parameters are:
  - pin: the Arduino pin number (2, 3)
  - ISR: the ISR to call when the interrupt occurs
  - mode: defines when the interrupt should be triggered. Four constants are predefined as valid values:
    - LOW to trigger the interrupt whenever the pin is low
    - CHANGE to trigger the interrupt whenever the pin changes value
    - RISING to trigger when the pin goes from low to high
    - FALLING for when the pin goes from high to low



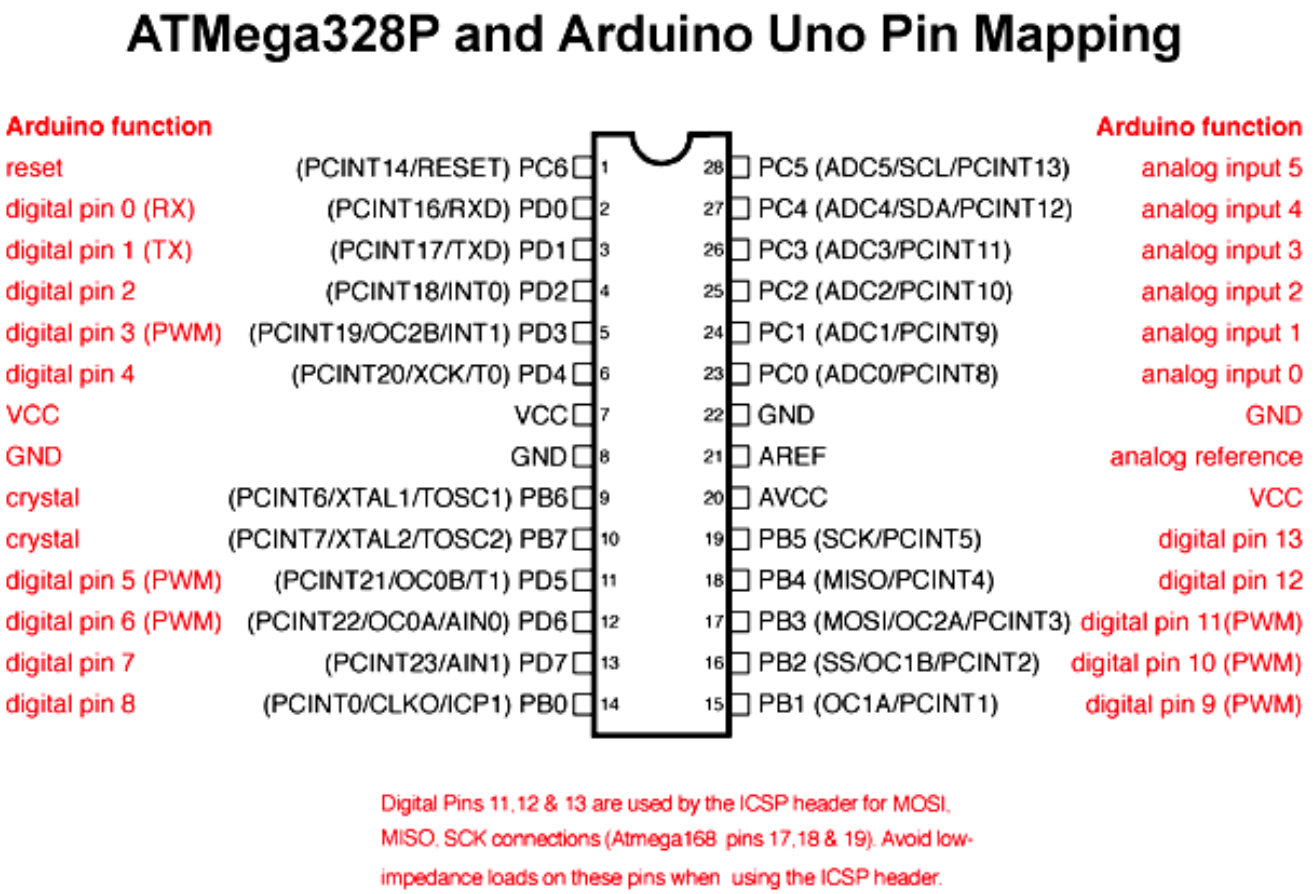
# Example button press code

- Easy to implement a simple button press routine
- Connect the push button to pin 2 and ground
- The global interrupt , GICR and MCUCR are all set using the attachInterrupt function
- With this code, the LED will turn on when the button is pressed and off when released
- Challenge: Can we think of a way to create a code for a button that performs different tasks if short-pressed or long-pressed?

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}

void loop() {
  digitalWrite(ledPin, state);
}

void blink() {
  state = !state;
}
```

# Internal interrupts

- Apart from the external pin based interrupts, we have many internal interrupts in most controllers
- In Arduino, the total number of internal interrupts is around 20 (including timer based interrupts)
- Some of the most used are (in order of decreasing priority):
  - Reset – external interrupt with no RELI()
  - INT0,1 – external pins
  - PCINT – external pins
  - Watchdog timer-out
  - ADC conversion complete
  - Analog comparator



# Internal interrupts

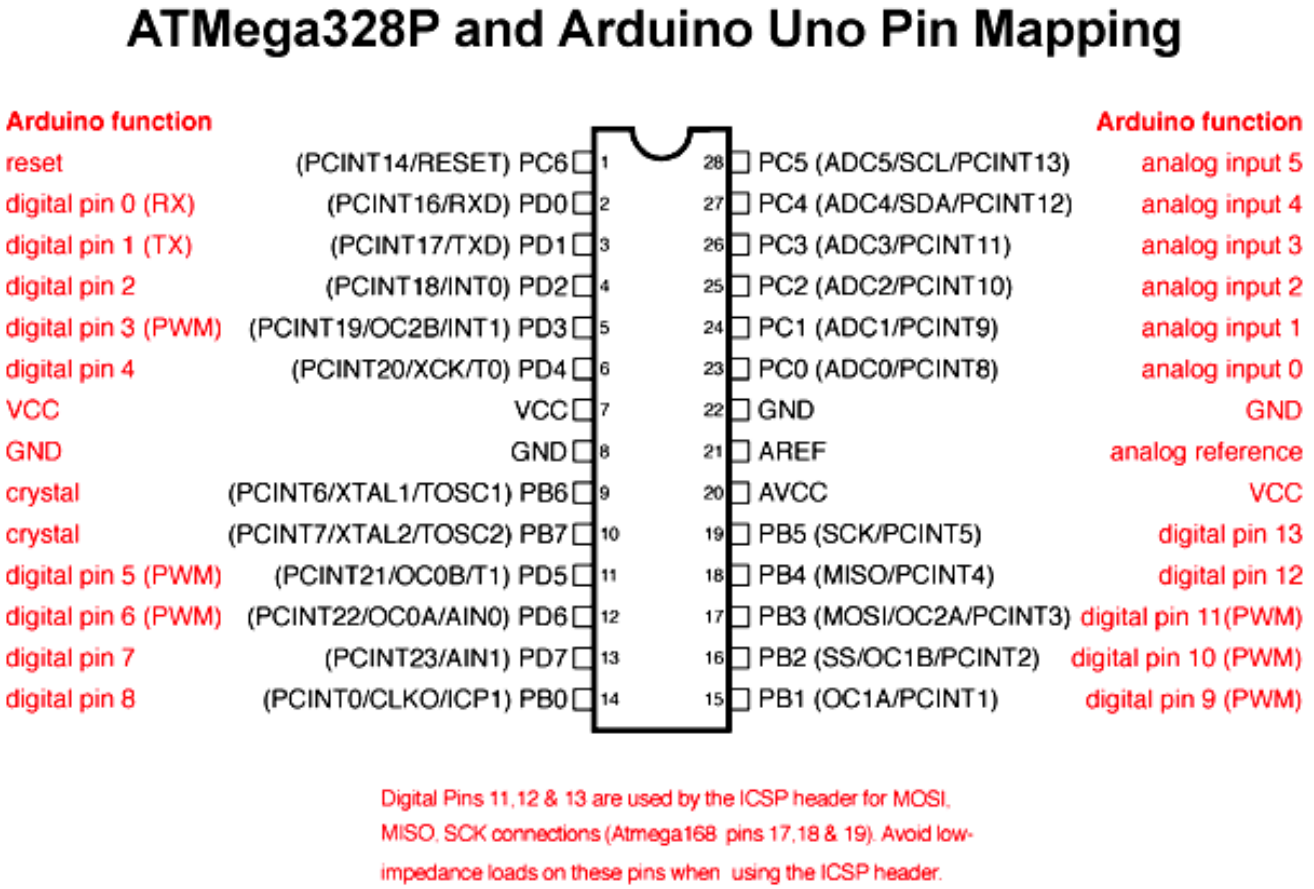
- ADC conversion complete
  - This interrupt signals that conversion from analog to digital is complete and the result is available as analogRead
  - Useful for sampling fast changing analog signals at the fastest rate possible
  - To enable this, we need to set the ADCSR register with bits like
    - ADEN – enable ADC
    - ADIE – enable ADC interrupt
  - The ADIF (ADC interrupt flag) is enabled when the conversion is over and the data register is updated – this is only done when ADIE and global interrupt is set
- Once done, the ISR can be programmed as:

```
ISR (ADC_vect)
{
//code
}
```

# Internal interrupts

- Analog comparator
  - This interrupt is triggered when the comparison output between pin AIN1 and AIN0 toggles in a particular direction
  - The incoming voltage is on the AIN0 (positive) pin which is pin 12 on the actual chip, and D6 on the Arduino board. The reference voltage (negative) pin is pin 13 on the chip, and D7 on the Arduino
  - To enable this, we set the ACSR:
  - ACIE – Analog Comparator Interrupt Enable
  - ACIS1 – Analog Comparator Interrupt Mode Select
- ACI - Analog Comparator Interrupt Flag is set when triggered

```
ISR (ANALOG_COMP_vect)
 {
 //code
 }
```



# Multiple interrupts

- There are cases when multiple interrupt flags will be enabled simultaneously
- In this case, the interrupt with the highest priority gets serviced
- The priority is determined beforehand in the boot-loader or can be software set in more advanced processors
- By default, interrupts are disabled during an ISR, however, there are controllers that “remember” the interrupts that got triggered during an ISR and service them once the ISR is done

| 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 |
| 19 | 0x0024 | USART, RX | USART Rx Complete |
| 20 | 0x0026 | USART, UDRE | USART, Data Register Empty |
| 21 | 0x0028 | USART, TX | USART, Tx Complete |
| 22 | 0x002A | ADC | ADC Conversion Complete |
| 23 | 0x002C | EE READY | EEPROM Ready |
| 24 | 0x002E | ANALOG COMP | Analog Comparator |
| 25 | 0x0030 | TWI | 2-wire Serial Interface |
| 26 | 0x0032 | SPM READY | Store Program Memory Ready |

# Nested interrupts

- In certain cases, interrupts can be enabled during the servicing of an ISR
- Thus, when a particular ISR is being processed, an interrupt can trigger and cause the program execution to go to that ISR
- This may also lead to recursive interrupts if permitted by the processor architecture
- This method is generally not recommended and should be avoided unless absolutely necessary
- Simple solutions like have multiple short ISRs is recommended

PATRIoT lab

# Timers

Dr. Aftab M. Hussain,

Assistant Professor, PATRIoT Lab, CVEST

# Timers

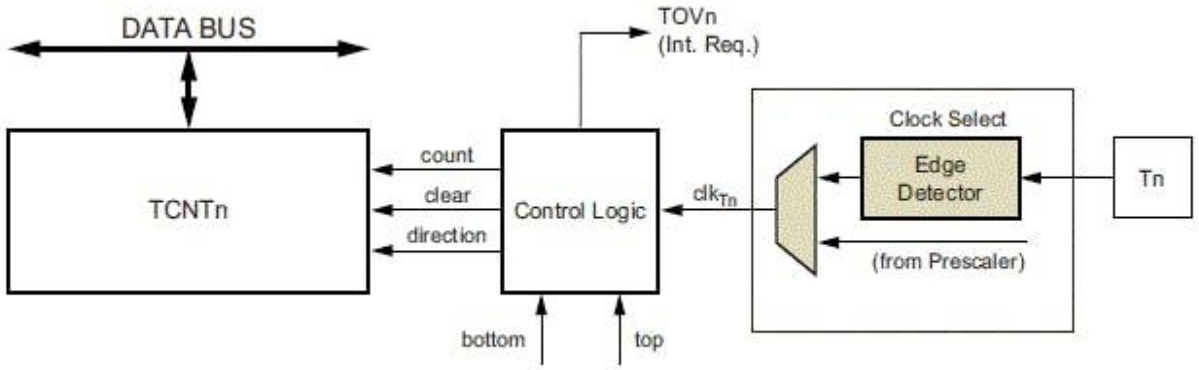
- Internal timers are used to calculate time using the known clock frequency obtained from the PLL output
- In essence, timers are just counters being provided with a particular clock frequency
- Once a particular count is over, a timer generally creates an interrupt that can trigger a particular application or even other timers to register their ticks
- Many Arduino functions uses timers, for example the time functions:
  - delay()
  - millis()
  - micros()
  - delayMicroseconds()

# Timers

- A delay() function is most commonly used to stall the program for a fixed amount of time
  - delay(x) stalls the program for x millisecond
- However, during this time, the program is not able to perform any other task
- Say a blinking LED is required while checking for an analog input
- If this is done using a delay() function, the delay function can inhibit the analogRead function

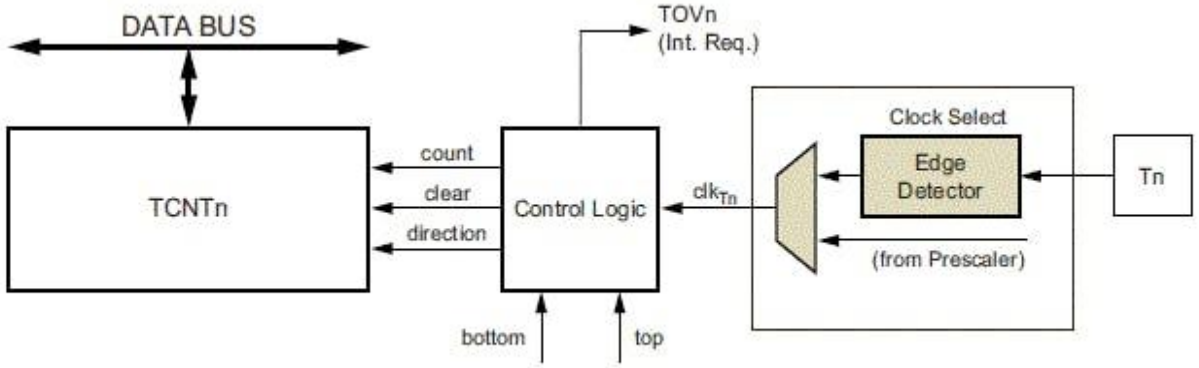
# Timers

- Another way of making the same code is using internal timers
- In Arduino, we have 3 timers, called timer0, timer1 and timer2
- Timer0 and timer2 are 8bit timers, where timer1 is a 16bit timer
- Thus, maximum count for timer0 and timer2 is 256 while that for timer1 is 65536
- These timers are implemented using TCNTn counter connected to the databus with a prescaled clock applied to each



# Timers

- All the timer values can be compared with a given value to generate an interrupt at a predefined time
- The registers involved in this are:
  - TCNTn - Timer/Counter Register. The actual timer value is stored here
  - TCCRn – Timer/Counter Control Register. Determines the settings for the timer
  - OCRn - Output Compare Register
  - TIMSKn - Timer/Counter Interrupt Mask Register. To enable/disable timer interrupts
  - TIFRn - Timer/Counter Interrupt Flag Register. Indicates a pending timer interrupt

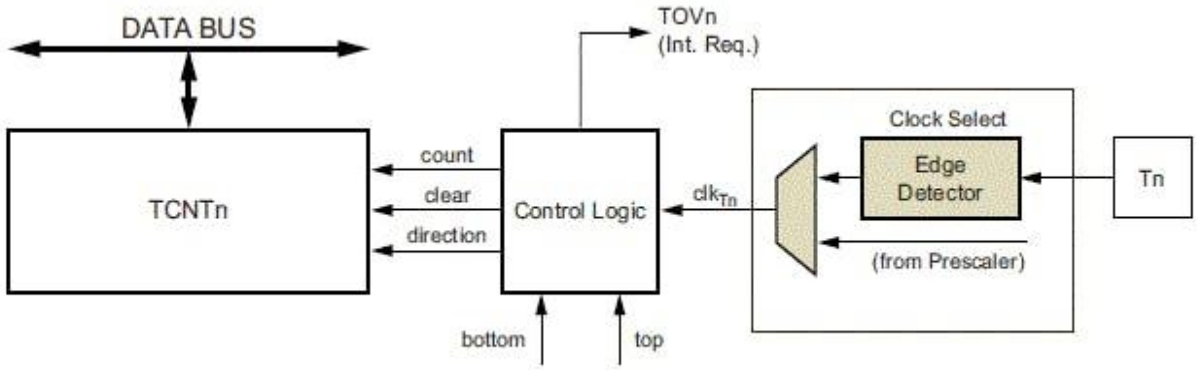


# Timers

- Apart from this, the prescaler needs to be set to scale the internal clock of the controller as needed

- This increases the range of the timer, however, decreases the precision with which the timing is measured

- For instance, if the clock of the controller is 16 MHz and the prescalar is 8, the actual clock applied to the timer is 16/8 = 2 MHz



# Timers

- Let us take the same example of an LED blinking at 0.5Hz
- To calculate the timer frequency you use the CPU frequency as 16Mhz for Arduino
- If we use timer0, the maximum timer counter is 256
- We set the TCCR0B to a particular prescalar using CS00, CS01 and CS02, say 64
- The clock frequency for timer is: 16 M/ 64 = 250000 Hz
- We cannot get the 1 sec clock with this, but we can get 1 ms clock
- If we need 1000Hz operation, we divide result through the desired frequency: 250000 / 1000Hz = 250
- Thus, the OCR0A value should be 249 (result – 1)

| | 7 bit | 6 bit | 5 bit | 4 bit | 3 bit | 2 bit | 1 bit | 0 bit |
|---|---|---|---|---|---|---|---|---|
| **TCCR0B** | FOC0A | FOC0B | - | - | WGM02 | CS02 | CS01 | CS00 |

*Timer/Counter Control Register 0 B*

| CS02 | CS01 | CS00 | DESCRIPTION |
|---|---|---|---|
| 0 | 0 | 0 | Timer/Counter0 Disabled |
| 0 | 0 | 1 | No Prescaling |
| 0 | 1 | 0 | Clock / 8 |
| 0 | 1 | 1 | Clock / 64 |
| 1 | 0 | 0 | Clock / 256 |
| 1 | 0 | 1 | Clock / 1024 |

# Timers

- An important function of all the three timers is the Clear Timer on Compare match
- When the timer counter (TCNTn) reaches the compare match register (OCRn), the timer will be cleared to zero
- This helps the timer frequency stay at the desired value
- This mode is enabled by setting the WGM02 bit in the TCCRnB register

| | 7 bit | 6 bit | 5 bit | 4 bit | 3 bit | 2 bit | 1 bit | 0 bit |
|---|---|---|---|---|---|---|---|---|
| **TCCR0B** | FOC0A | FOC0B | - | - | WGM02 | CS02 | CS01 | CS00 |

*Timer/Counter Control Register 0 B*

| CS02 | CS01 | CS00 | DESCRIPTION |
|---|---|---|---|
| 0 | 0 | 0 | Timer/Counter0 Disabled |
| 0 | 0 | 1 | No Prescaling |
| 0 | 1 | 0 | Clock / 8 |
| 0 | 1 | 1 | Clock / 64 |
| 1 | 0 | 0 | Clock / 256 |
| 1 | 0 | 1 | Clock / 1024 |

# Timers

```
/*
This program turns on and off a LED on pin 13 each 1 second
*/

int timer=0;
bool state=0;
void setup() {
  pinMode(13,OUTPUT);

  TCCR0A=(1<<WGM01);    //Set the CTC mode
  OCR0A=0xF9; //Value for ORC0A for 1ms

  TIMSK0|=(1<<OCIE0A);   //Set the interrupt request
  sei(); //Enable interrupt

  TCCR0B|=(1<<CS01);    //Set the prescale 1/64 clock
  TCCR0B|=(1<<CS00);
```

```
void loop() {
  //in this way you can count 1 second because the nterrupt
  if(timer>=1000){
    state=!state;
    timer=0;
  }

  digitalWrite(13,state);

}

ISR(TIMER0_COMPA_vect){    //This is the interrupt request
  timer++;
}
```

# Internal timer implementations – millis()

- Measuring a time period using millis, is simply a matter of comparing current time to the time value stored in a variable. As you go round a loop you continuously perform a simple bit of maths: millis() - stored_time
- This gives you the elapsed time in milliseconds
- The millis() function is driven by a millisecond timer interrupt that increments an unsigned long every time it activates and just returns the value of that variable
- This is implemented internally using timer0 (64 bit prescaler)

```
LOOP
  ...
  // An event happens
  if (event==true) stored_time = millis();
  ...
  elapsed_time = millis() - stored_time;
  ...
END_LOOP
```

# Internal timer implementations – millis()

- The millis value is stored in an unsigned int of 32 bit
- Thus, it can store a maximum value of 4,294,967,296
- This translates to roughly 49.71 days before it overflows back to 0
- It is always prudent to define the variable that is supposed to contain millis values as unit32_t
- millis can also be used to create non-blocking delays

```
void setup (void) {
}

#define LED 13

void loop(void){
static uint8_t tog=0;
static uint32_t oldtime=millis();

    if ( (millis()-oldtime) > 500) {
       oldtime = millis();

       tog = ~tog; // Invert
       if (tog) digitalWrite(LED,HIGH); else digitalWrite(LED,LOW);

    }
}
```

# Watchdog timers

- The ATmega328P has a Watchdog Timer which is a useful feature to help the system recover from scenarios where the system hangs or freezes due to errors in the code written or due to conditions that may arise due to hardware issues
- The watchdog timer uses an internal 128kHz clock source
- When enabled, it starts counting from 0 to a value selected by the user
- If the watchdog timer is not reset by the time it reaches the user selected value, the watchdog resets the microcontroller
- This hard reset is used to make sure the controller does not get stuck in infinite loops
  - Particularly helpful in IoT applications where wifi and lorawan connectivity is needed

# Watchdog timers

- Configure the watchdog timer through one of the registers known as WDTCSR
  - WDIE - Enables Interrupts. This will give you the chance to include one last dying wish before the board is reset. This is a great way of performing interrupts on a regular interval should the watchdog be configured to not reset on time-out
  - WDE - Enables system reset on time-out. Whenever the Watchdog timer times out the micocontroller will be reset. This is probably what you were all looking for. Set this to ‘1’ to activate
  - WDP0/WDP1/WDP2/WDP3 - These four bits determine how long the timer will count for before resetting

| Bit | Name |
|---|---|
| 7 | WDIF |
| 6 | WDIE |
| 5 | WDP3 |
| 4 | WDCE |
| 3 | WDE |
| 2 | WDP2 |
| 1 | WDP1 |
| 0 | WDP0 |

| WDP 3 | WDP 2 | WDP 1 | WDP 0 | Time-out (ms) |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 16 |
| 0 | 0 | 0 | 1 | 32 |
| 0 | 0 | 1 | 0 | 64 |
| 0 | 0 | 1 | 1 | 125 |
| 0 | 1 | 0 | 0 | 250 |
| 0 | 1 | 0 | 1 | 500 |
| 0 | 1 | 1 | 0 | 1000 |
| 0 | 1 | 1 | 1 | 2000 |
| 1 | 0 | 0 | 0 | 4000 |
| 1 | 0 | 0 | 1 | 8000 |

# Watchdog timers

- To make life easy, Arduino implementations have some functions to configure the WDTCSR
- We have:

wdt_enable(WDTO_8S)
options: WDTO_1S, WDTO_2S, WDTO_4S, WDTO_8S

wdt_disable()

wdt_reset()

```
#include<avr/wdt.h> /* Header for watchdog timers in AVR */

void setup() {
  Serial.begin(9600); /* Define baud rate for serial communication */
  Serial.println("Watchdog Demo Starting");
  pinMode(13, OUTPUT);
  wdt_enable(WDTO_2S);  /* Enable the watchdog with a timeout of 2 seconds */
}

void loop() {
  for(int i = 0; i<20; i++) /* Blink LED for some time */
  {
    digitalWrite(13, HIGH);
    delay(100);
    digitalWrite(13, LOW);
    delay(100);
    wdt_reset();  /* Reset the watchdog */
  }
  while(1); /* Infinite loop. Will cause watchdog timeout and system reset. */
}
```

# Watchdog timers

- To let the watchdog timer know that everything is running ok and that it needn't panic or take any action you are going to have to keep resetting the timer within your main loop

- This is done by periodically entering in:

  wdt_reset();

```
#include<avr/wdt.h> /* Header for watchdog timers in AVR */

void setup() {
  Serial.begin(9600); /* Define baud rate for serial communication */
  Serial.println("Watchdog Demo Starting");
  pinMode(13, OUTPUT);
  wdt_enable(WDTO_2S);  /* Enable the watchdog with a timeout of 2 seconds */
}

void loop() {
  for(int i = 0; i<20; i++) /* Blink LED for some time */
  {
    digitalWrite(13, HIGH);
    delay(100);
    digitalWrite(13, LOW);
    delay(100);
    wdt_reset();  /* Reset the watchdog */
  }
  while(1); /* Infinite loop. Will cause watchdog timeout and system reset. */
}
```

# Summary

- We discussed many important features of hardware design using microcontrollers
- The specific register names and bit positions will be different for different controllers. Timer sizes and clock frequencies will be different
- However, the basic concepts of interrupts, timers, watchdogs are generally the same
- These are very helpful features in advanced IoT applications