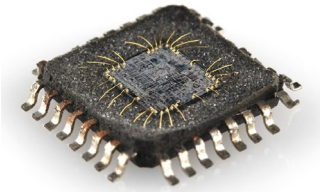
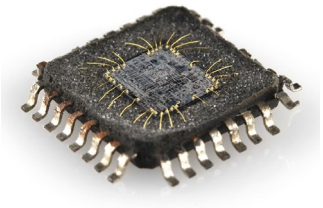
Agenda

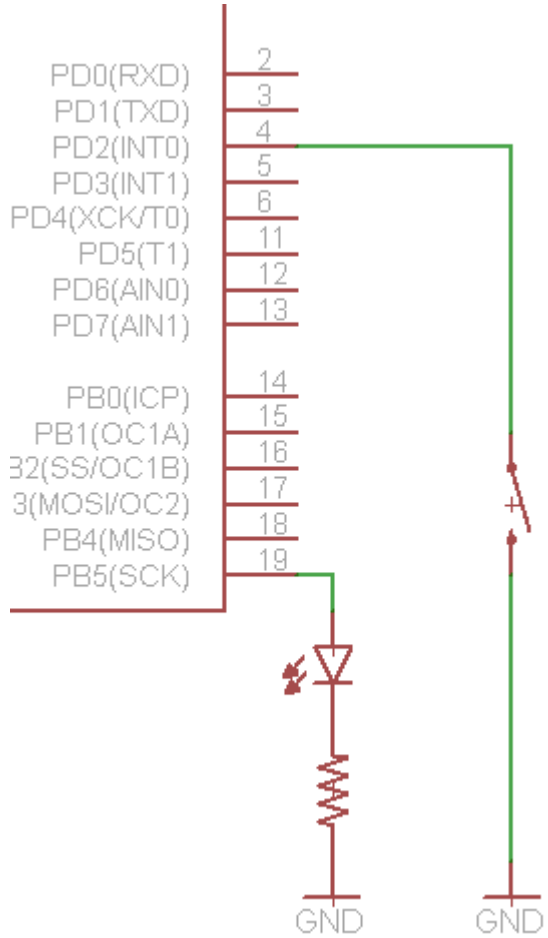


1. Interrupts
2. Timers

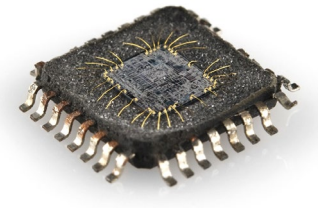
Call Back: Button as an input



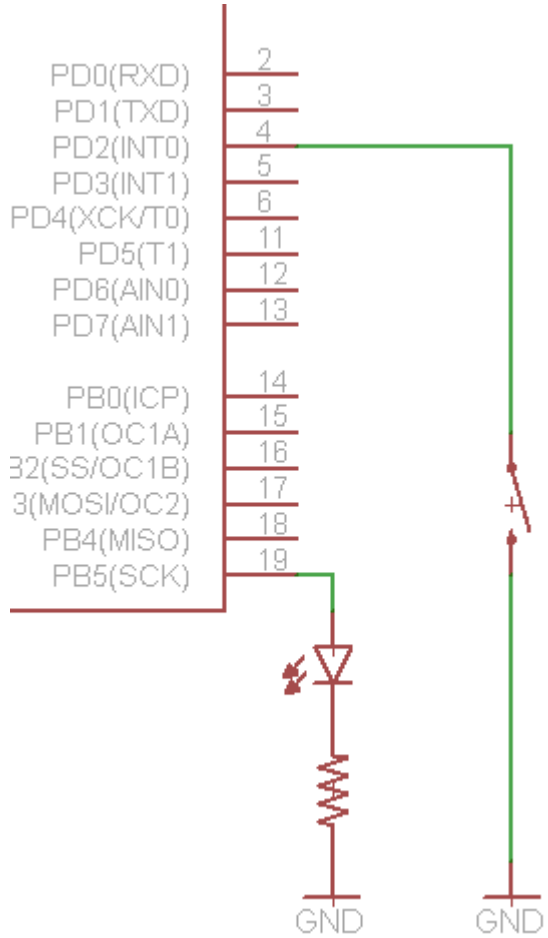
```
int buttonPin = 2;
int LED = 13;

void setup() {
   pinMode(buttonPin, INPUT_PULLUP);
   pinMode(LED, OUTPUT);
}

void loop(){
   int buttonValue = digitalRead(buttonPin);

   if (buttonValue == LOW){
    digitalWrite(LED,LOW);
   } else {
    digitalWrite(LED, HIGH);
   }
}
```



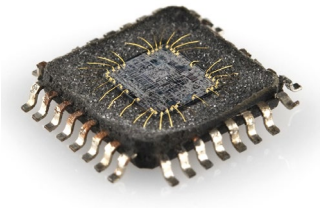
Rewrite #1: Button as an input



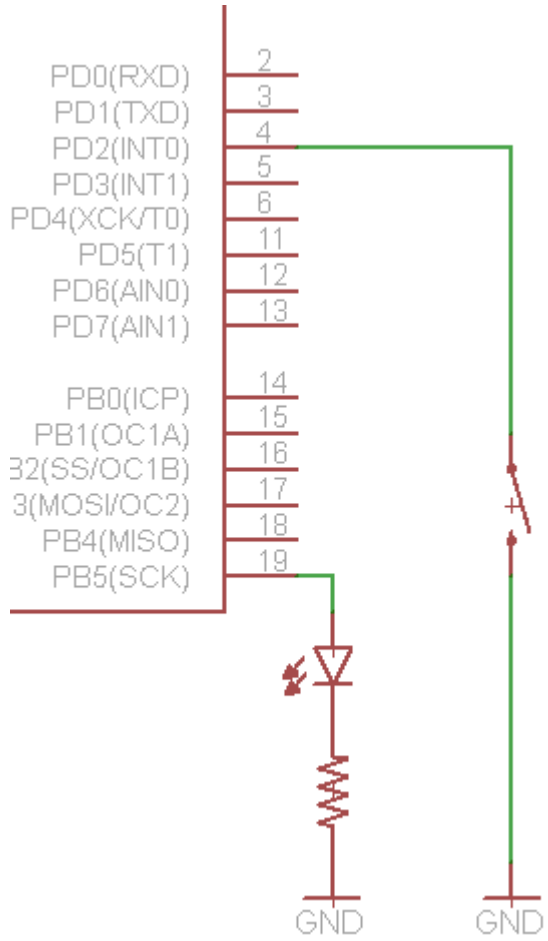
```
const byte LED = 13, SW = 2;

void setup() {
   pinMode(SW, INPUT_PULLUP);
   pinMode(LED, OUTPUT);
}

void handleSW() {
    digitalWrite(LED, digitalRead(SW));
}

void loop() {
    handleSW();
}
```



Rewrite #2: Button as an input

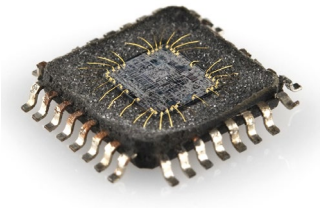


```
const byte LED = 13, SW = 2;

void handleSW() {
    digitalWrite(LED, digitalRead(SW));
}
void handleOtherStuff() {
    delay(250);
}

void setup() {
   pinMode(SW, INPUT_PULLUP);
    pinMode(LED, OUTPUT);
}

void loop() {
    handleSW();
    handleOtherStuff();
}
```



- Anything Wrong with this Example ?

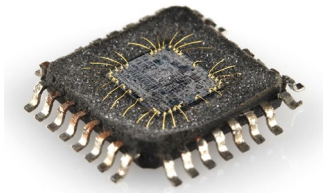
Rewrite #2: Button as an input



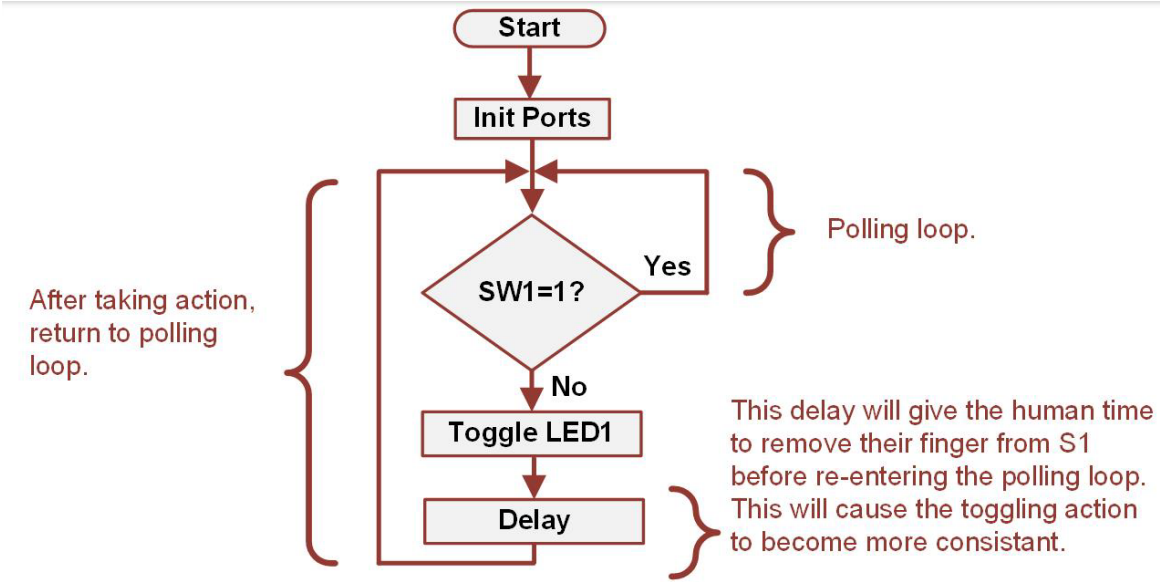
```
const byte LED = 13, SW = 2;

void handleSW() {
    digitalWrite(LED, digitalRead(SW));
}
void handleOtherStuff() {
    delay(250);
}

void setup() {
   pinMode(SW, INPUT_PULLUP);
   pinMode(LED, OUTPUT);
}

void loop() {
    handleSW();
    handleOtherStuff();
}
```

- This code implements a classic strategy known as polling.
- Sometime polling is the only way to get something done but it should generally be avoided, if possible, for some equally classic reasons:
  1. First and foremost, if the statements in *handleSW* block take much time, then it will take that much time before we check the digital IO pin again.
  2. Or the user may have pressed and released the button while the microcontroller is dutifully executing its way through other work (*handleOtherStuff*) meaning the button press will be missed.
  3. Alternatively, if the user didn't press the button, then the microcontroller will busily check the digital pin wasting power if running on a battery.

Rewrite #2: Button as an input



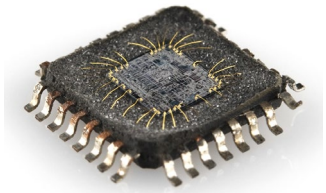
- The Drawback of Polling – CPU spends a lot of time executing instructions that did nothing but actively check for an infrequent event.



- Interrupt (IRQ) – an approach to dealing with external, asynchronous events by building hardware interrupts om the MCU that handles identifying and prioritizing events to be serviced by the CPU; only reacts one time when a transition is observed.

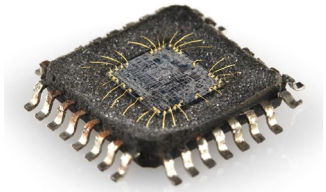
Interrupts

Why Interrupts?



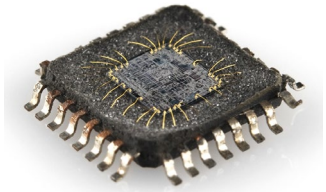
- Consider the following situation:
  - You are working at your desk at home on your assignment.
  - The phone rings (an interrupt).
  - You stop work on your assignment and answer the phone (you accept the interrupt). It is your friend, who wants to know the name and phone number of your auto mechanic so she can call to arrange a service for her car.
  - You give her the name and number (you process the interrupt request immediately).
  - You then hang up and go back to work on your taxes.
- The additional time it will take you to complete your taxes is minuscule, yet the amount of time for your friend to complete her task may be significantly reduced.
- This simple example clearly illustrates how interrupts can **drastically improve response time in a real-time system**.
- Using interrupts requires that we first understand how the CPU processes an interrupt so that we can select the most appropriate software solution for each I/O device.

Why Interrupts?



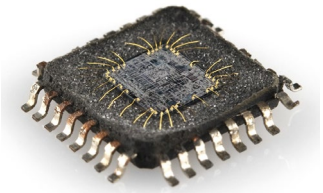
- Interrupts have a reputation for being tricky and an unnecessary complication to just get code running.
- This reputation is however ill-deserved.
- Interrupts when used **correctly** it can:
  1. Simplify code
  2. Speed execution on hardware
  3. Allow projects to do more with less powerful hardware

What is an Interrupt?

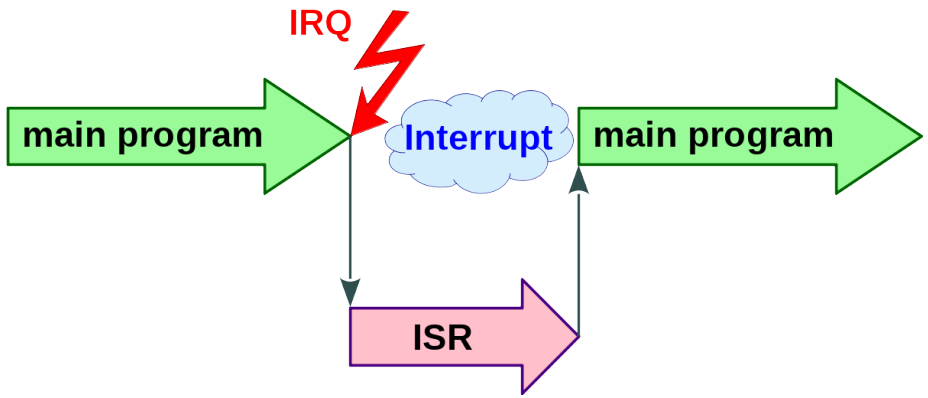


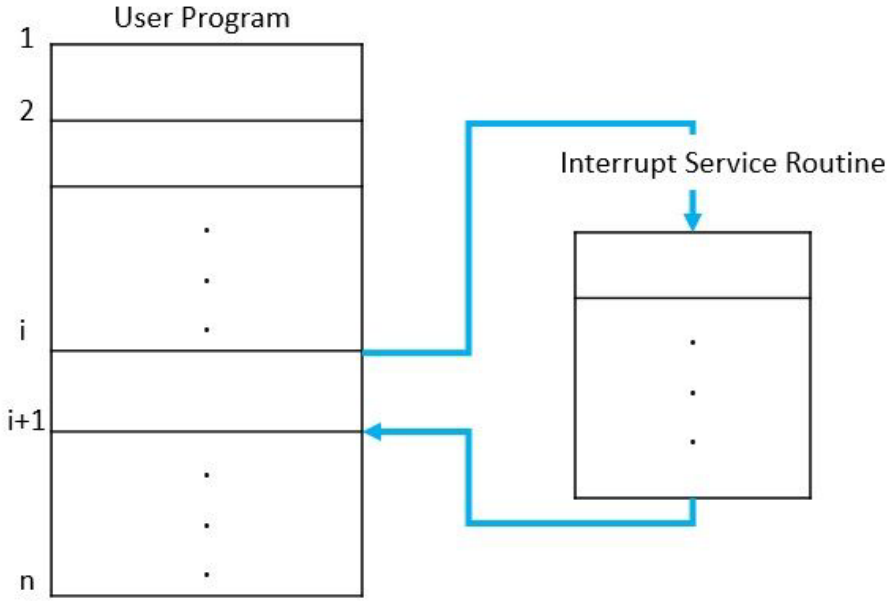
- Normally a microcontroller executes a set of instructions, your code, in a sequence.
- Some of those instructions are what we call flow control or branching instructions which cause execution to take different paths, think if-else statement blocks.
- These instructions, even the branching instructions, are the normal flow of your program.
- An interrupt stops the microcontroller from following the normal flow and causes it to execute a different block of code, typically resuming the normal flow of your code once done.

What is an Interrupt?

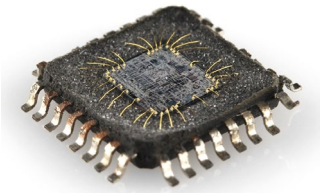


- Interrupts are a way for a microcontroller to temporarily stop what it is doing to handle another task.
- The currently executing program is paused, an **ISR (interrupt service routine)** is executed, and then your program continues, none the wiser.

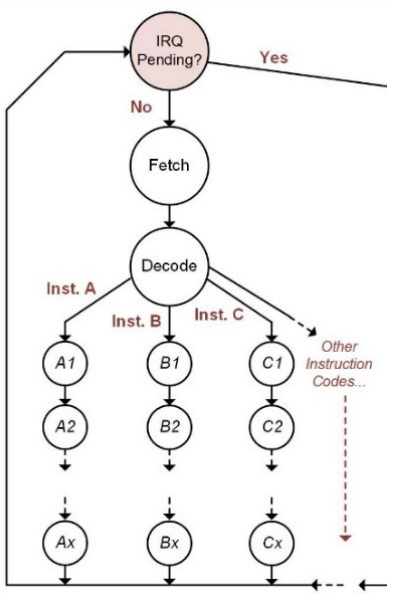


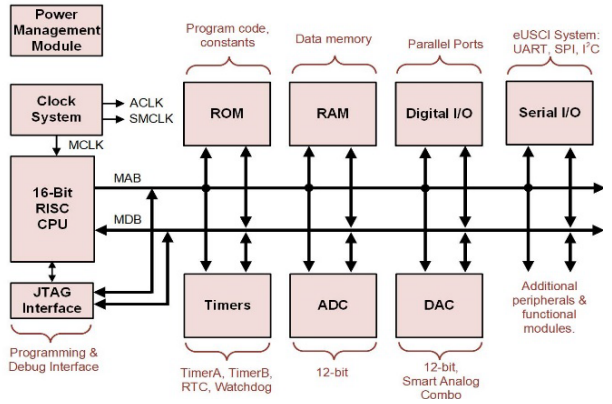


Interrupt Flags (IFG)

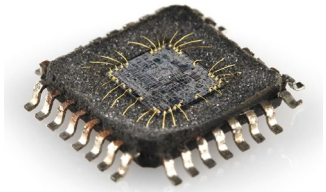


- **Flag** – notifies the CPU that an external event on a peripheral has occurred and action is requested.
- The term interrupt stems from the fact that the CPU takes a break from executing the main program and instead executes instructions specifically for the peripheral event.
- This process is highly efficient because the CPU does not have to spend execution cycles polling each external system.



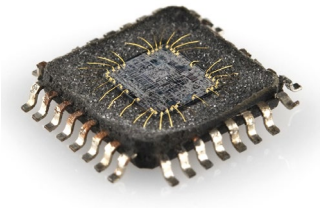


When would you use one?

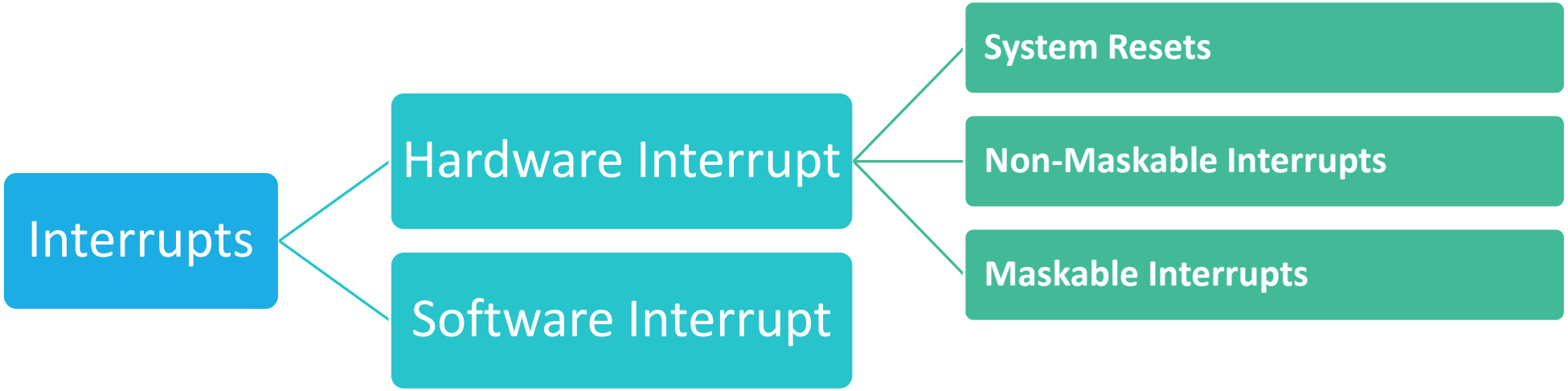


- Interrupts can detect brief pulses on input pins. Polling may miss the pulse while you are doing other calculations.
- Interrupts are useful for waking a sleeping processor.
- Interrupts can be generated at a fixed interval for repetitive processing.
- And more …

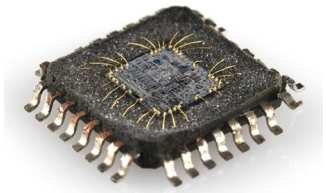
Types of Interrupts



- There are two types of interrupts:
  1. **Hardware Interrupts** - are meanwhile triggered by hardware; whether internal such as a real time clock indicating a second has elapsed or external via a signal on an interrupt capable digital IO pin. Such as in response to an external event, like a pin going high or low.
  2. **Software Interrupts** - These occur in response to a software instruction. such as an error like division by zero.



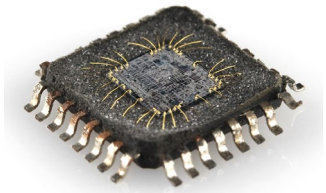
Interrupt Priority



1. System resets
2. Non-maskable interrupts (NMIs)
3. Maskable interrupts

- **System Resets** – highest priority interrupts and are always enabled; causes the MCU to start its operation from the beginning.
- System resets include power-on reset (POR), power-up reset (PUR), external reset, and power supply monitor violation.
- The only action needed by developer for system resets is **to tell the interrupt system where the starting address of the main program is**.

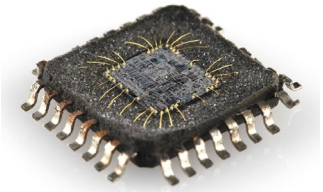
Interrupt Priority



1. System resets
2. **Non-maskable interrupts (NMIs)**
3. Maskable interrupts

- **Non-maskable Interrupts** – second highest priority interrupts; typically handle fault conditions on the MCU.
- Examples include memory access errors and oscillator faults.
- Non-maskable interrupts are always enabled but are different from system resets in that they do **execute developer written ISRs** instead of a set of predetermined actions.

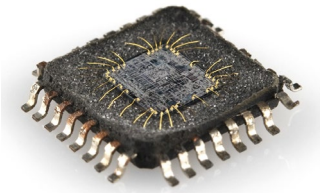
Interrupt Priority



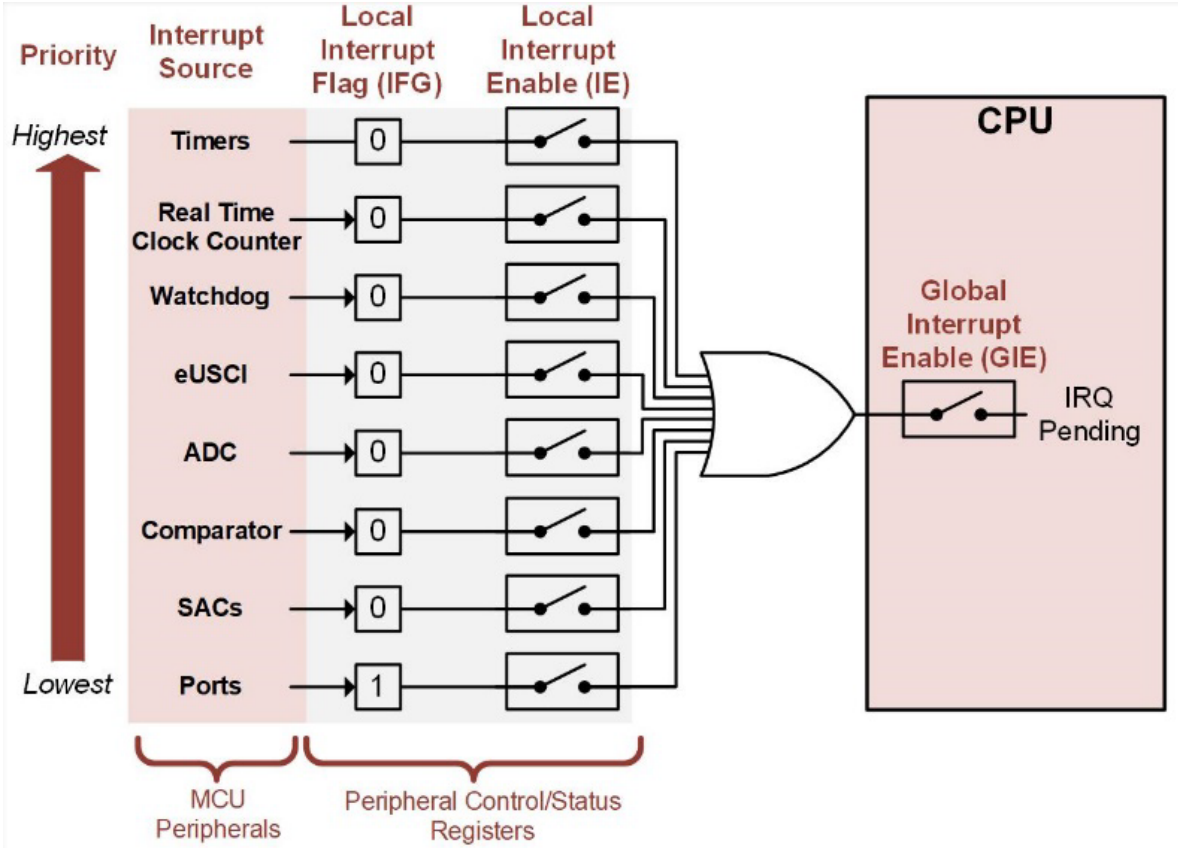
1. System resets
2. Non-maskable interrupts (NMIs)
3. **Maskable interrupts**

- **Maskable Interrupts** – third highest priority interrupts.
- These are the peripherals:
  - Ports
  - Timers
  - Serial interface
  - ADC
  - DAC

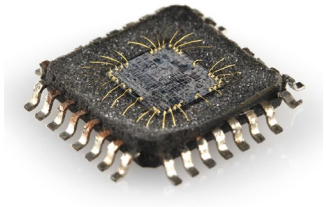
Interrupt Priority



- Maskable Interrupts – means that they can be turned on or off.

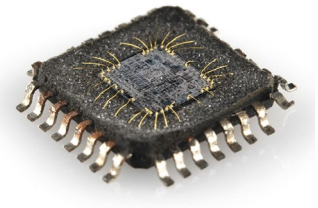


Types of Interrupts

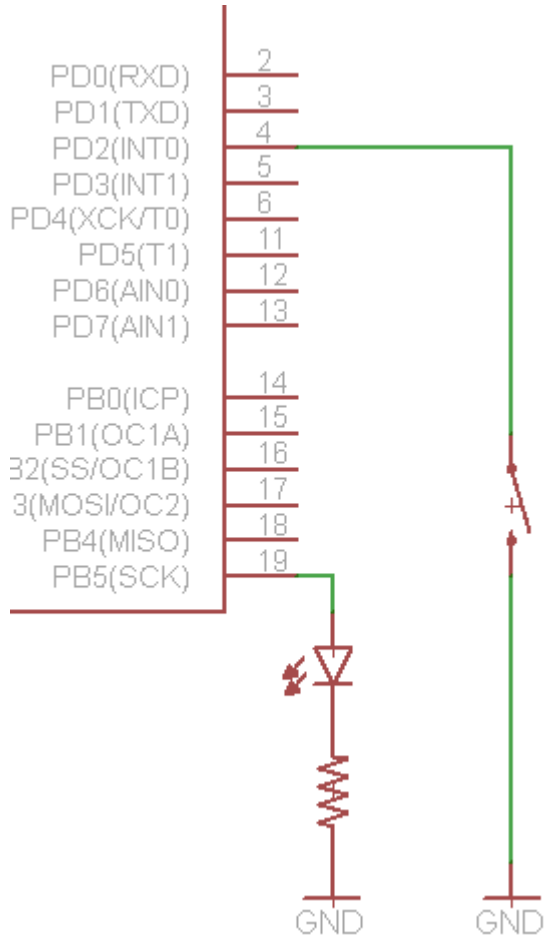


- Generally, most 8-bit AVR microcontrollers (i.e., Arduinos or ATMega328) **aren't innately capable of software interrupts**, so, we will focus on hardware interrupts.
- There are 26 different interrupts on an **Arduino Uno**
  - 1 Reset
  - 2 External Interrupt Request 0 (pin D2)
  - 3 External Interrupt Request 1 (pin D3)
  - 4 Pin Change Interrupt Request 0 (pins D8 to D13)
  - 5 Pin Change Interrupt Request 1 (pins A0 to A5)
  - 6 Pin Change Interrupt Request 2 (pins D0 to D7)
  - 7 Watchdog Time-out Interrupt
  - 8 Timer/Counter2 Compare Match A
  - …
  - 18 SPI Serial Transfer Complete
  - 19 USART Rx Complete
  - …
  - 25 2-wire Serial Interface (I2C)
  - …

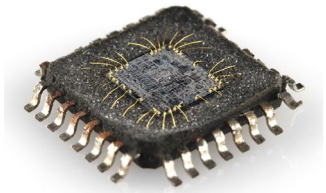
Rewrite #3 “Interrupt”: Button as an input



```
const byte LED = 13, SW = 2;

void handleSW() {  // now it's our ISR  ←
    digitalWrite(LED, digitalRead(SW));
}
void handleOtherStuff() {
    delay(250);
}

void setup() {
    pinMode(SW, INPUT_PULLUP);
    pinMode(LED, OUTPUT);
    attachInterrupt(INT0, handleSW, CHANGE);   ←
}

void loop() {
    // handleSW();          ← commented out
    handleOtherStuff();
}
```



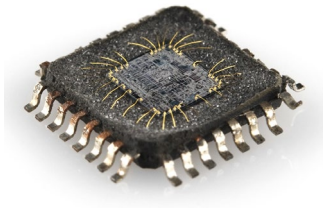
ISR



- Interrupt Service Routines should be kept **short**.
- Interrupts are disabled when the ISR is called, so other interrupts are **postponed**.
- Do not call millis() or delay() or Serial or …
- This one is good:

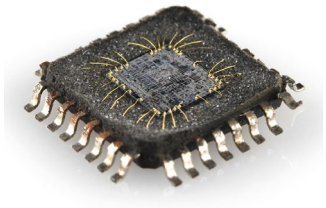
```
void myISR () {
    count++;
}
```

Interrupt Modes



- 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
  - HIGH to trigger the interrupt whenever the pin is high

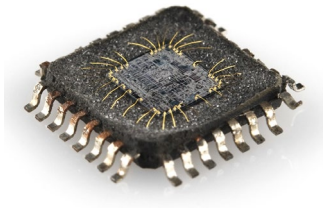
Example: Sharing Data



```
const byte LED = 13, SW = 2;
volatile unsigned char count = 0;
unsigned char lastCount = -1;


void handleSW() {
   digitalWrite(LED, digitalRead(SW));
   count++;
}

void handleOtherStuff() {
    if (count != lastCount) {
        Serial.print("Count ");
        Serial.println(count);
        lastCount = count;
    }
}
```

```
void setup() {
    //Start up the serial port
    Serial.begin(9600);
    Serial.println(F("Tracking ISR"));

   pinMode(SW, INPUT_PULLUP);
   pinMode(LED, OUTPUT);
   attachInterrupt(INT0, handleSW, CHANGE);
}

void loop() {
    // handleSW();
    handleOtherStuff();
}
```

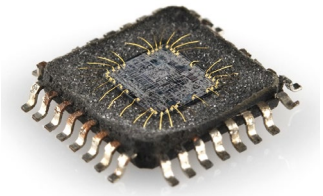
More on sharing data



- An interrupt can happen at any time.
- If you share a multi-byte value (e.g., short int) between an ISR and your code, you must take additional precautions.
- Since this is an 8-bit microcontroller, multibyte values take multiple instructions to fetch a value from memory.
- Consider what happens if count is 255 and you get an interrupt that increments count between instructions 1fa and 1fe.

```
volatile short count;
if (count == 256) ...

1fa:    80 91 10 01      lds r24, 0x0110 ; count lower
1fe:    90 91 11 01      lds r25, 0x0111 ; count upper
202:    80 50            subi    r24, 0x00
204:    91 40            sbci    r25, 0x01
206:    69 f5            brne    .+90
```

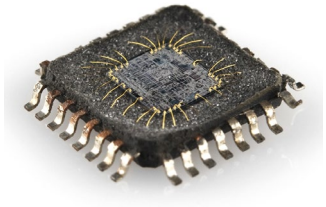
More on sharing data, cont'd



```
// Disable interrupts and copy
noInterrupts();
short int myCount = count;
interrupts();
if (myCount == 256) ...
```

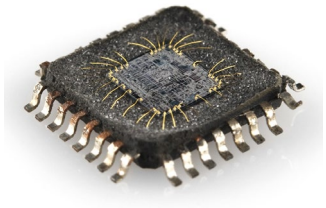
```
1fa:   f8 94           cli
1fc:   80 91 10 01     lds r24, 0x0110
200:   90 91 11 01     lds r25, 0x0111
204:   78 94           sei
206:   80 50           subi    r24, 0x00
208:   91 40           sbci    r25, 0x01
20a:   69 f5           brne    .+90
```

Pin Change Interrupt



- Pin 2 is INT0
- Pin 3 is INT1
- But, what about pins 0,1,4,5,6,…
- Pin Change Interrupts can monitor all pins

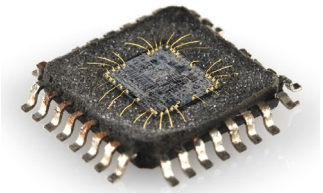
Example: PinChangeInt library



```
#include <PinChangeInt.h>

const byte LED = 13, SW = 5;
volatile unsigned char count = 0;
unsigned char lastCount = -1;

void handleSW() {
   digitalWrite(LED, digitalRead(SW));
   count++;
}

void handleOtherStuff() {
    if (count != lastCount) {
        Serial.print("Count ");
        Serial.println(count);
        lastCount = count;
    }
}
```

```
void setup() {
    //Start up the serial port
    Serial.begin(9600);
    Serial.println(F("Tracking ISR"));

   pinMode(SW, INPUT_PULLUP);
   pinMode(LED, OUTPUT);
   PCintPort::attachInterrupt(SW, handleSW,
   CHANGE);
}

void loop() {
handleOtherStuff();
}
```

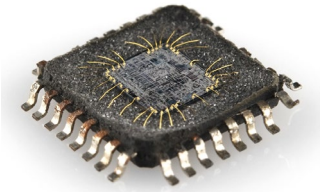
Example: wake from sleep



```
#include <avr/sleep.h>
#include <PinChangeInt.h>

void wake() { // ISR
    sleep_disable();                    // first thing after waking from sleep:
    PCintPort::detachInterrupt(SW);     // stop LOW interrupt
}

void sleepNow() {
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);
    noInterrupts();                     // stop interrupts
    sleep_enable();                     // enables sleep bit in MCUCR
    PCintPort::attachInterrupt(SW, wake, LOW);
    interrupts();                       // allow interrupts
    sleep_cpu();                        // here the device is put to sleep
}
```

Timers

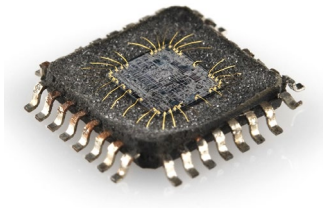
Call Back: Blinking



- How the **Microcontroller** handles the delay function ?

```
void setup() {
   pinMode(LED_BUILTIN, OUTPUT);
}
void loop() {
     digitalWrite(LED_BUILTIN, HIGH);
     delay(500);
     digitalWrite(LED_BUILTIN, LOW);
     delay(500);
}
```

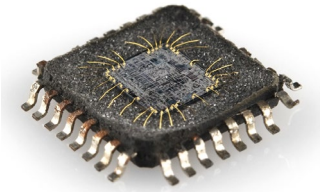
Example: Blinking without Blocking



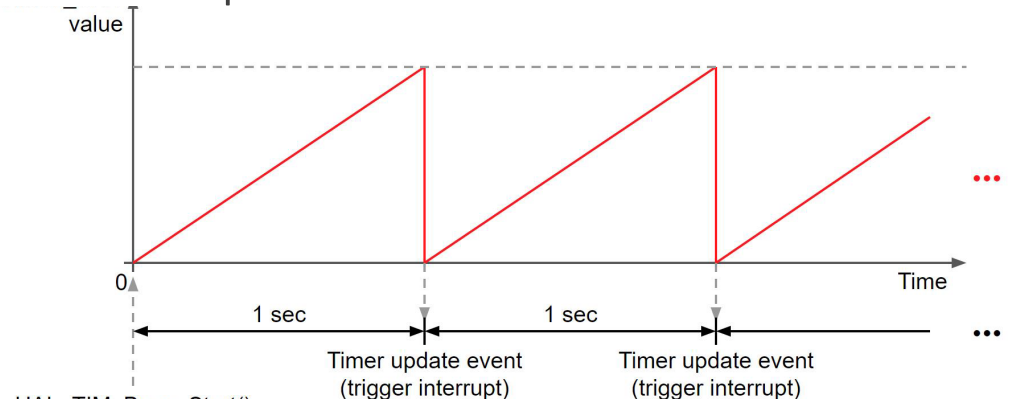
```
int previousLEDstate = LOW;                    // for keeping the previous LED state
unsigned long lastTime = 0;                    // Last time the LED changed state
int interval = 500;                            // interval between the blinks in milliseconds

void setup() {
    pinMode(LED_BUILTIN, OUTPUT);              // Declare the pin for the LED as Output
}
void loop(){
    unsigned long currentTime = millis();      // Read the current time
    if (currentTime - lastTime >= interval){   // Compare the current time with the last time
      lastTime = currentTime;                  // First we set the previous time to the current time

      if (previousLEDstate == HIGH) {          // Then we inverse the state of the LED
         digitalWrite(LED_BUILTIN, LOW);
         previousLEDstate = LOW;
      } else {
         digitalWrite(LED_BUILTIN, HIGH);
         previousLEDstate = HIGH;
      }
    }
```

- Can you rewrite the loop function in a single line of code ?

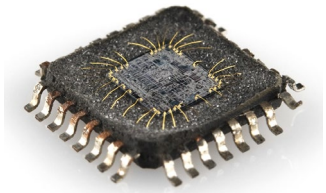
What is a timer?



- Timers are an important part of the functionalities in microcontrollers and play a vital role in controlling various aspects of it.
- A **timer** *or* **counter** is a piece of hardware built in the microcontroller (such as Arduino, AVR, ...). It is like a clock and can be used to measure time events or carry out specific tasks at a particular interval of time..
- Imagine our timer as a counter that counts pulses.
- The timer is fed by the system clocl

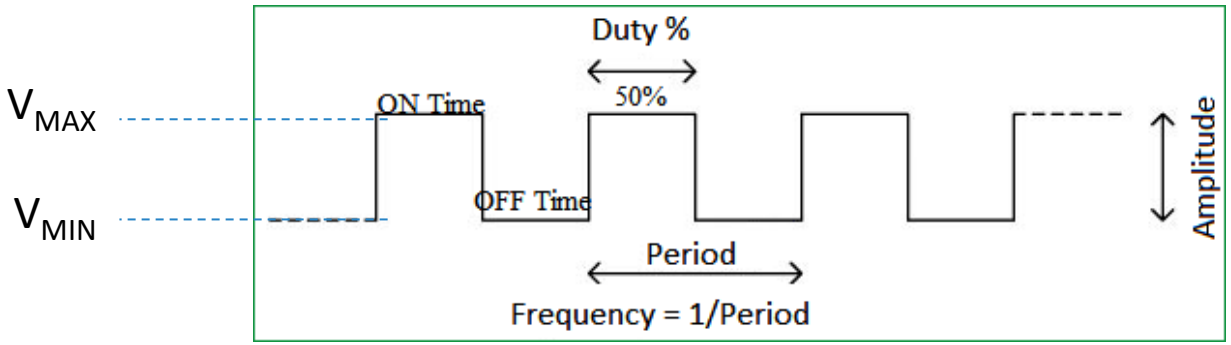


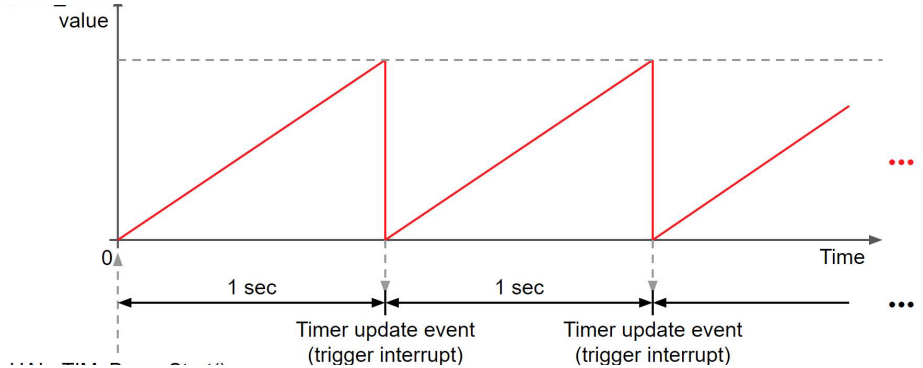
Applications of the timers

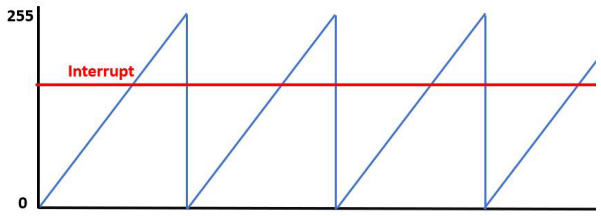


1. **PWM — Pulse Width Modulation** is a method to control the duty cycle, thereby the digital output of the controller.
   It has a wide range of applications from motor control to LED dimming and much more.

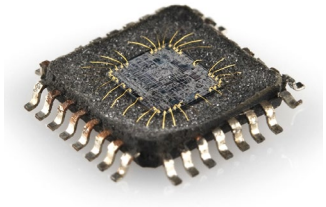
2. **Timer Interrupts** — In a lot of cases we need to count the exact time before an event occurs or give a desired output of a specific pin after some time





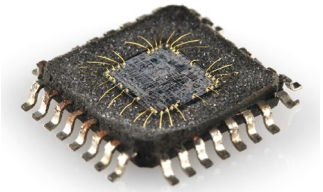


Timers in Arduino



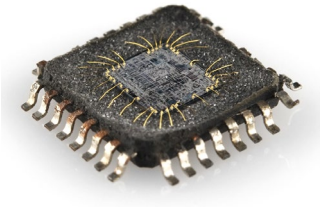
- Depending on each microcontroller, there are multiple timers available to work with.
- Every timer has a counter which increments the timer on each tick of the timer's clock.
- As Arduino programmer you will have used timers and interrupts without knowledge, bcause all the low-level hardware stuff is hidden by the Arduino API.
- Many Arduino functions uses timers, for example the time functions:
  - delay(), millis() and micros() and delayMicroseconds().
  - The PWM functions analogWrite() uses timers, as the d() and the noTone() function does.
  - Even the Servo library uses timers and interrupts.

Timers in Arduino



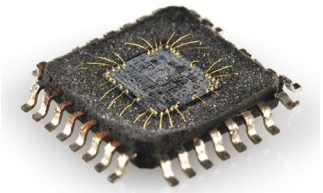
- The controller of the Arduino Uno is the Atmel AVR ATmega328 or Atmega168.
- These chips are pin compatible and only differ in the size of internal memory. Both have 3 timers:
  1. Timer0 – 8 bit – used by delay(), millis(), micros(), controls PWM on pins 5 and 6.
  2. Timer1 – 16 bit – used by Servo library, controls PWM on pins 9 and 10.
  3. Timer2 – 8 bit – used by tone(), controls PWM on pins 11 and 3.
- The most important difference between 8bit and 16bit timer is the timer resolution.
  - 8bits means 256 values
  - where 16bit means 65536 values for higher resolution.

Timers in Arduino

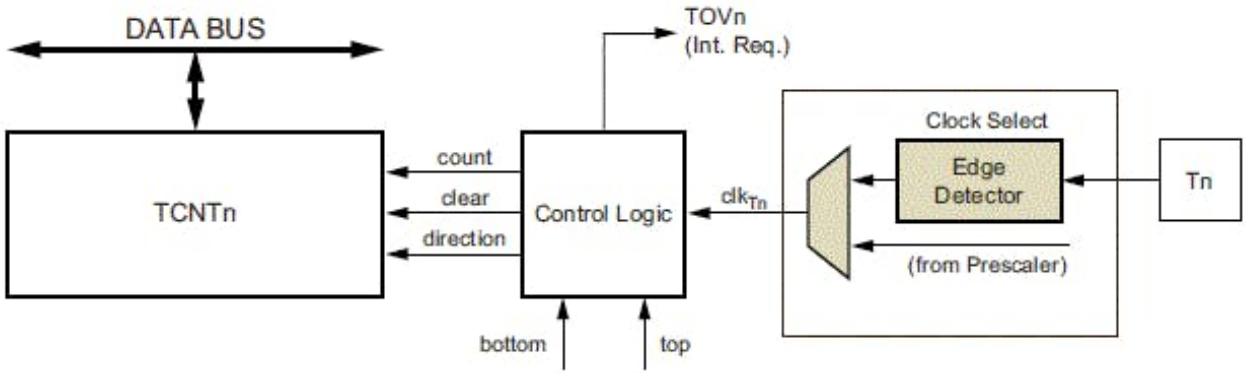


- The microcontroller for the Arduino Mega series is the Atmel AVR ATmega1280 or the ATmega2560.
- They are also identical only differs in memory size. These controllers have 6 timers:
  - Timer 0, timer1 and timer2 are identical to the ATmega168/328.
  - The timer3, timer4 and timer5 are all 16bit timers, similar to timer1.
- All timers depends on the system clock of your Microcontroller or Arduino system.

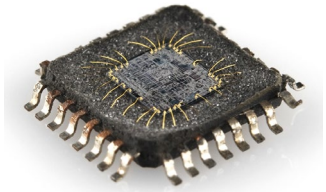
Timers in Arduino



- A 16Mhz clock acts as a base clock in the AVR ATmega328 (Arduino Uno)
- but 16Mhz is too fast for our application, we can divide it by number in order to make it feasible for our use which is known as the Prescaler.
- Prescalar is a piece of hardware that divides the number of pulses from the system (base) clock by the number you define such as 8, 64, 256 and so on...



Timers in Arduino

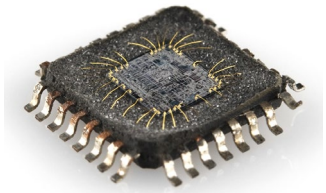


- A prescaler dictates the speed of your timer according the the following equation:

$$Timer\ Speed\ (Hz) = \frac{Base\ Clock\ Speed\ (Hz)}{Prescaler}$$

- For Example, 16Mhz crystal that will create a 16Mhz square signal. That's 62.5ns for each pulse, right?
- If prescalar is set to 8, the output signal from the prescalar will be 8 time slower so it will be 500ns for each pulse.
- So, each 8 pulses from the system clock, our timer will increase by 1.

Timer interrupts



1. Output Compart Match (or Compare Match)

   We can write a value in a different register and when the timer value is equal to the compare value, it will trigger the interrupt.

   For example, we set our compare match register to 100, each time timer 0 reaches 100, it will create an interruption

2. Overflow Interrupt

   In this case an interruption is triggered each time the timer overflows, meaning it passes from its maximum value back to 0,

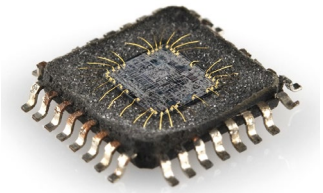
   For example, which in case of an 8-bit timer will be each time it reaches 255.

3. Timer Input Capture

   In this case the timer could store its value (timestamp) in a register, each time an external event happens on one of the input pins.

   It will also set a flag indicating that an input has been captured. This allows the system to continue executing without interruption while an input is being received while still having the capability to trigger events based on the exact time when the input was received.

Setting the Prescaler



- AVR ATmega328 use two main registers to control the timers, the **TCCRxA** and the **TCCRxB**, for each timer where x is the number of the timer. So, for timer 1 we have TCCR1A and TCCR1B.

TCCR1A – Timer/Counter1 Control Register A

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Compare Output Mode

Waveform Generation Mode (PWM)

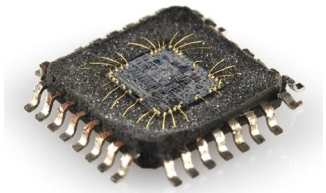
TCCR1B – Timer/Counter1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Input Capture

Clock Select

Setting the Prescaler



TCCR1A – Timer/Counter1 Control Register A

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	–	–	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCCR1B – Timer/Counter1 Control Register B

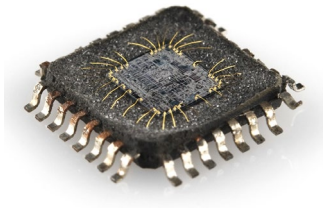
Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 15-6. Clock Select Bit Description

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (no prescaling)
0	1	0	$clk_{I/O}$/8 (from prescaler)
0	1	1	$clk_{I/O}$/64 (from prescaler)
1	0	0	$clk_{I/O}$/256 (from prescaler)
1	0	1	$clk_{I/O}$/1024 (from prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

- all we care are **the first 3 bits** (CS10, CS11 and CS12 bits) which are used to define the prescaler value.
- we can disable the prescaler or set it to 1, divided by 8, 64, 256, 1024 or even use an external clock source.
- For the timer 0 you must use the TCCR0B and bits CS00, CS01, CS02.
- For the timer 2 you must use the TCCR2B and bits CS20, CS21 and CS22.

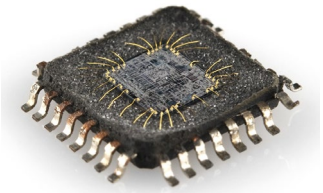
Setting the Prescaler



```
void setup() {
  TCCR1A = 0;                 //Reset entire TCCR1A register
  TCCR1B = 0;                 //Reset entire TCCR1B register
  TCCR1B |= 0x04; // B00000100;  //Set CS12 to 1 so we get Prescalar = 256
  TCNT1 = 0;                  //Reset Timer 1 value to 0
}

void loop() {
  //your code here...
}
```

- The timer value is stored in the **TCNT** register which in case of timer 1 is made out of two registers because it is a 16 bits one. So, if you want to reset the timer value you should equal the TCNT register to 0
- Why we use this prescaler then?

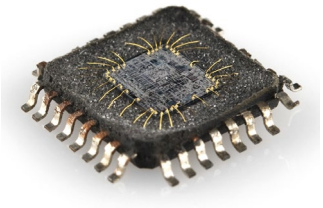
Setting the Prescaler



- Well, Imagine you want to make an LED blink each half second. So, you want to trigger the timer interruption each 500ms, right?
- The system clock is 16Mhz, so each pulse is 62.5ns. In order to count up to 500ms, which are 5,000,000ns we would need to count up to 8,000,000 so we can't use the 16 bit register for example, because it could only count up to 65,536.
- But if we use a prescaler of 256, we will have a pulse each 16µs. So now, to count up top 500ms we would need 500.000 divided by 16 equal to 31,250 pulses, and now we could use the 16 bits register.

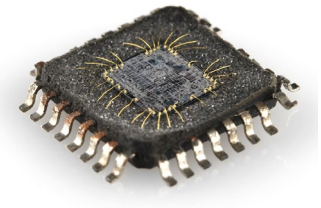
  To set timer 1 to have a 256 prescalar, according to the table above, we need to set the CS10, CS11 and VS12 to be 1, 0, 0.

Example: Blinking using Timers



```
/* Blinking every 500ms using timer1 and prescaler of 256.Calculations (for 500ms) */
bool LED_STATE = true;
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);

  cli();                                //stop interrupts for till we make the settings
  TCCR1A = 0;                           // Reset entire TCCR1A to 0
  TCCR1B = 0;                           // Reset entire TCCR1B to 0

  TCCR1B |= 0x04;                       //Set CS12 to 1 so we get prescalar 256
  TIMSK1 |= 0x02;                       // B00000010; enable compare match mode on register A by Set OCIE1A to 1
  OCR1A = 31250;                        //Finally we set compare register A to 31250
  sei();                                //Enable back the interrupts
}

void loop() { }

// With the settings above, this IRS will trigger each 500ms.
ISR(TIMER1_COMPA_vect){
  TCNT1  = 0;                           //First, set the timer back to 0 so it resets for next interrupt
  LED_STATE = !LED_STATE;               //Invert LED state
  digitalWrite(LED_BUILTIN,LED_STATE);              //Write new state to the LED on pin D5
}
```

Example: Blinking using Timers



```
/* Blinking every 500ms using timer1 and prescaler of 256.Calculations (for 500ms) */
bool LED_STATE = true;
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);

  TCCR1A = 0;                         // Reset entire TCCR1A to 0
  TCCR1B = 0;                         // Reset entire TCCR1B to 0

  TCCR1B |= 0x04;                     // bitSet(TCCR1B, CS12);
  TIMSK1 |= 0x02;                     //  bitSet(TIMSK1, TOIE1);
}

ISR(TIMER1_OVF_vect) {
  LED_STATE = !LED_STATE;             //Invert LED state
  digitalWrite(LED_BUILTIN,LED_STATE);              //Write new state to the LED on pin D5

}
void loop() { }
```

- Can we rewrite this without LED_STATE?

THANK YOU