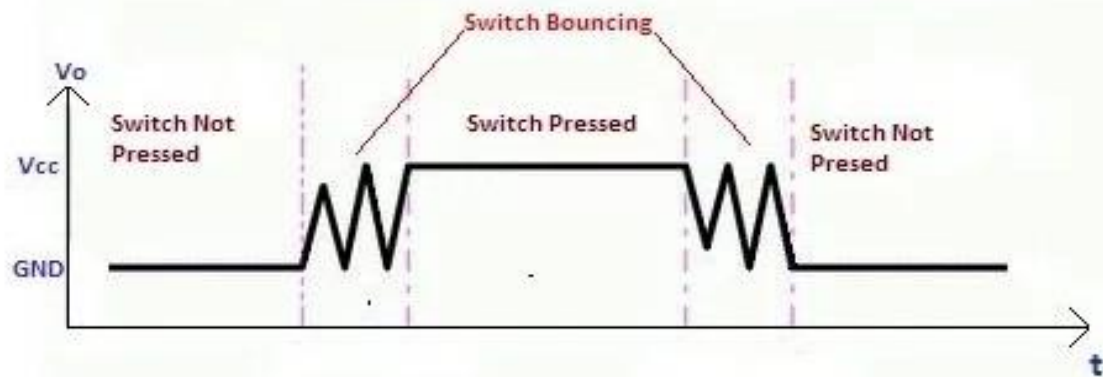
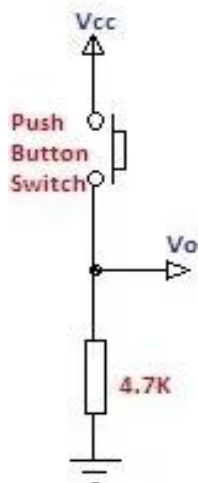


# Switch Debouncing and Interrupts

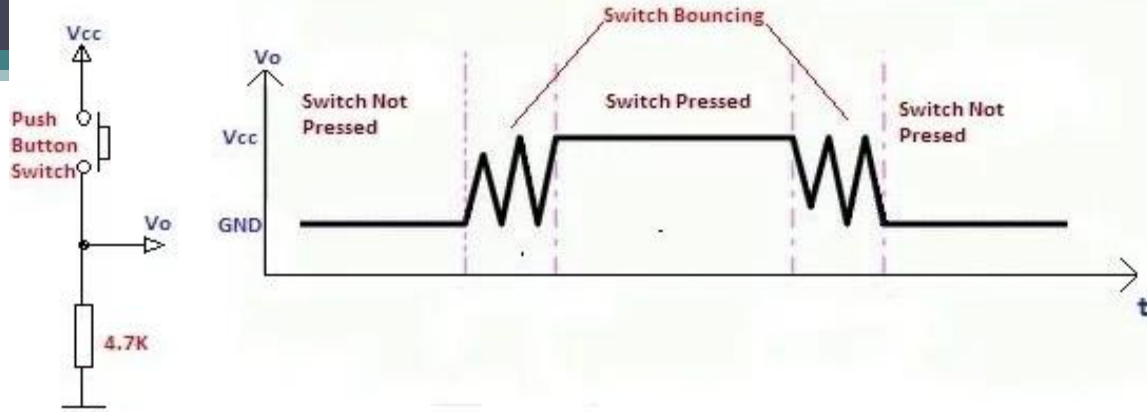
A series of horizontal lines in teal and light blue colors, with varying lengths and offsets, creating a modern, layered effect across the width of the slide.

# What is Debouncing?

- When the state of a **mechanical switch** is changed from **open to closed** and vice versa, there is often a short period of time when the input **voltage will jump between high and low levels a couple of times** before it settles at the applied value.
- These **transitions** are called **bouncing**, and getting rid of the effect of the transitions is called **debouncing**.
- If you want to input a manual switch signal into a digital circuit, you will need to **debounce the signal**, so a single press doesn't appear like multiple presses.



# Explanation



- Bouncing is a result of the **physical property of mechanical switches**.
- Switch and relay contacts are usually made of **springy** metals so when a **switch** is pressed, its essentially two metal parts coming together. This does not happen immediately, the switch bouncing between **in-contact** and **not-in-contact** until it finally settles down.
- When **people push** a mechanical switch button, they **expect one reaction** per push. As the buttons tend to **bounce around** when pressed and released, this will **mess up the signal** from them.
- For **example**, let's say we have a button that is connected between a voltage supply and the output probe. We intend to get an output of **5V (logic 1)** when the switch is pressed, and **0V (logic 0)** when unpressed. If we probed the output signal coming from the button during the transition from pushing it down to letting go, we expect an immediate and clean transition **0  $\rightarrow$  1  $\rightarrow$  0**. What we end up seeing instead is the picture above. Before the signal settles to a flat 5V, it bounces between the two logic states many times.

# Why we need debouncing?

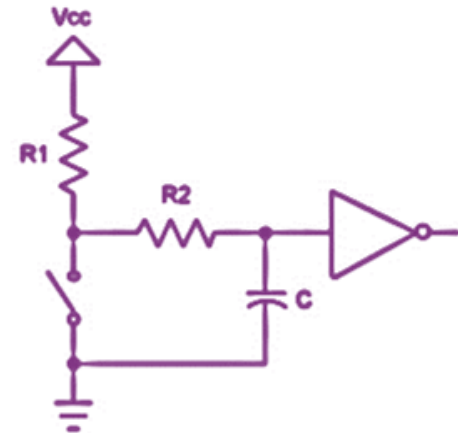
- Bouncing often causes problems in the application where, pressing the switch once should theoretically cause only one transition.
- **Example:** Imagine using a button in a TV remote for the selection of a channel. If the button is not being debounced, one press can cause the remote to skip one or more channels.

# Types of debouncing

- There are **two** general common way of getting rid of the bounces:
  - **Hardware debouncing:** adding **circuitry** that actually **gets rid of** the **unwanted transitions**.
  - **Software debouncing:** adding **code** that causes the application **to ignore** the **bounces**.

# Hardware Debouncing

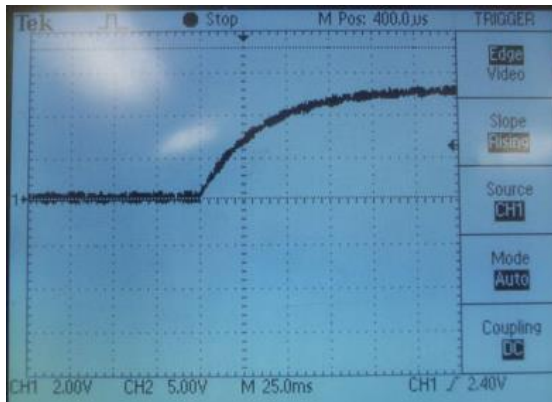
- There are **various implementations of circuits** that can be used for **eliminating the effect of switch debouncing** right at the hardware level.
- One of the most popular solutions using hardware is to employ a **resistor/capacitor network** with a **time constant ( $\tau$ )** longer than the **time it takes the bouncing contacts to stop**, i.e.,  $\tau > t_{bounce}$ . Often, it is also followed by a Schmitt trigger which helps get rid of the capacitor voltage values midway between **HIGH**
- If we wish to **preserve program execution cycles**, it is best to use the **hardware debouncing approach**.



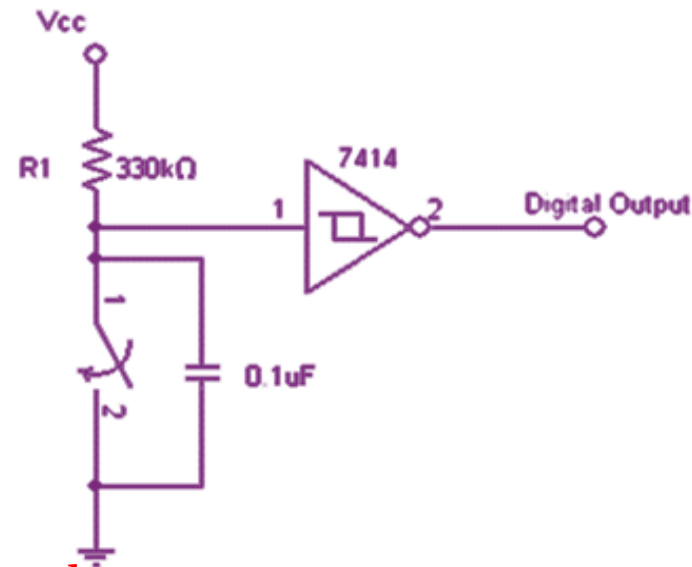
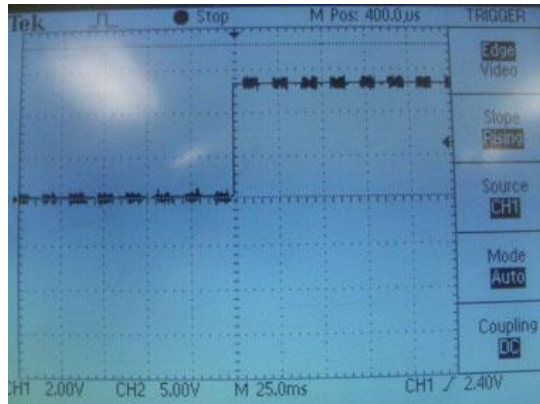
# Hardware Debouncing

There is one problem: the capacitor exhibits a curve when charging, and parts of this curve will fall in the region where a logic chip is undecided whether to treat it as a HIGH or LOW. To fix this, a **Schmitt trigger** is added to the output.

Without a Schmitt trigger:



With a Schmitt trigger:



Remember that the 7414 Schmitt trigger will invert the signal.

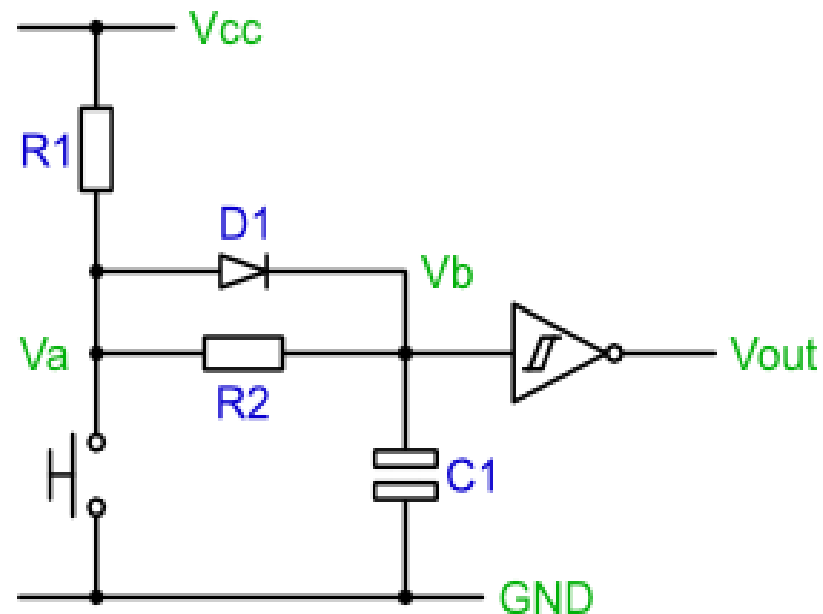
# Hardware Debouncing

Circuit Design using the 7414 is a **Schmitt trigger (inverting type buffer)** with the **input hysteresis**:

The equation for **discharging a capacitor** is given by-

$$v_c = v_{th} = V_{final} e^{-t/RC}$$

$V_{final}$  is the voltage from where the capacitor starts to discharge after the capacitor is completely charged.



Debouncing circuit design using the 7414 Schmitt trigger (inverting type buffer)

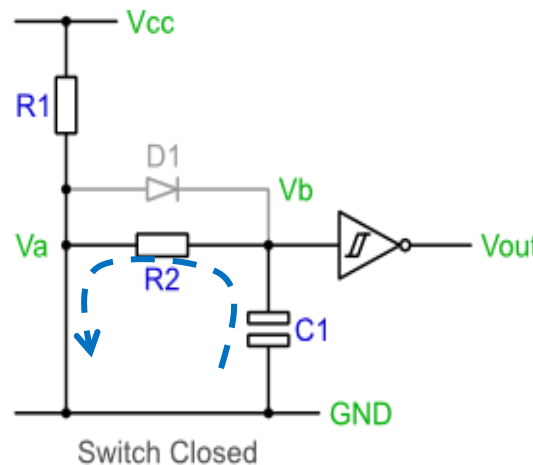
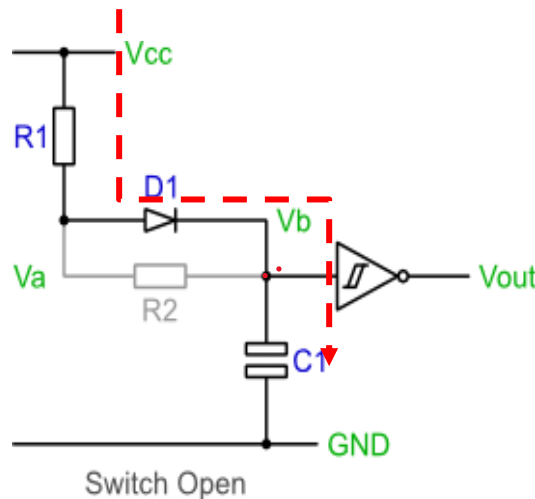


# Hardware Debouncing

The circuit's operation can be explained by looking at the equivalent circuits formed in the two switching states, e.g., **open** and **closed**.

**Starting with the switch OPEN:**

- The capacitor  $C_1$  will **charge** via  $R_1$  and  $D_1$ .
- In time,  $C_1$  will charge and  $V_b$  will reach within 0.7 V of  $V_{cc}$
- Therefore, the output of the **inverting Schmitt trigger** will be a **logic 0**.



Now, **CLOSE** the switch:

- The capacitor will **discharge** via  $R_2$ .
- In time,  $C_1$  will discharge and  $V_b$  will reach 0 V.
- Therefore, the output of the **inverting Schmitt trigger** will be a **logic 1**.

Remember that the 7414 Schmitt trigger will invert the signal.

# Hardware Debouncing

## What about the diode?

The resistor  $R_2$  is required as a discharge path for the capacitor. Without it,  $C_1$  will be shorted when the switch is closed. Without the diode,  $D_1$ , both  $R_1$  and  $R_2$  would form the capacitor's charging path when the switch is open. The combination of  $R_1$  and  $R_2$  increases the capacitor's charging time, and thus slows down the circuit. However, when the switch is closed,  $R_1$  is connected across the supply rails to limit the current flow to the circuit.

# Hardware Debouncing Example

## Numerical Example:

A CMOS device like the 74AHCT14 dribbles about a  $\mu\text{A}$  from the inputs.

Both  $R_1$  and  $R_2$  control the capacitor's charging time, and so set the **debounce period** for the **switch open state**.

## Solution:

The **equation for capacitor charging** is:

$$V_c = V_{final}(1 - e^{-t/RC})$$

Where,  $V_{final}$  is the voltage towards which the capacitor will reach after it is completely charged. This will be  $V_{CC}$  in this case.

# Hardware Debouncing Example

The goal is to select values that ensure the capacitor's voltage stays above the  $V_{th}$ , the threshold at which the gate switches till the switch stops bouncing.

Most of the switches exhibit bounce times well under 10 ms, so the use of this value would be a **conservative choice**. Now increase that by the bounce duty cycle. Based on experimental data, we can expect about a 50% duty cycle, giving us 20 ms of a time period.

Rearranging the **capacitor discharging formula** to solve for  $R$  (the cost and size of capacitors vary widely so it is the best option to select a value for  $C$  and then compute  $R$ ) yields:

$$v_c = v_{th} = V_{final} e^{-t/RC}$$

$$R = - \frac{t}{C \ln \frac{V_{th}}{V_{final}}}$$

# Hardware Debouncing Example

The AHCT version has a **worst-case  $V_{th}$**  for a signal going **low of 1.7 V (in this case, it is the higher/upper threshold voltage,  $V_{th,H}$ )**.

Let us try with a 0.1  $\mu\text{F}$  capacitor, since this is small, cheap, and solve for the conditions in which the **switch is closed**. The **capacitor discharges** through  $R_2$ . If the power supply is 5 V (i.e.,  $V_{final} = 5 \text{ V}$ ) then

$$R_2 = \frac{20 \times 10^{-3}}{0.1 \times 10^{-6} \ln \frac{1.7}{5}} = 185 \text{ k}\Omega.$$

Since a resistor with this value is not available, so we use 180 k $\Omega$ .

But the analysis ignores the gate's input leakage current. A CMOS device like 74AHCT14 dribbles about 1  $\mu\text{A}$  from the inputs. That 180 k $\Omega$  resistor will bias the input up to 0.18 V ( $V = IR = 1 \times 10^{-6} \times 180 \times 10^3 = 0.18 \text{ V}$ ), uncomfortably close to the gate's best-case **switching point of 0.5 V**. Therefore, change the capacitor's capacitance to 1  $\mu\text{F}$  and use 18 k $\Omega$  for  $R_2$ .

# Hardware Debouncing Example

Now, we should compute the value of  $R_1$  during charging time.  $R_1 + R_2$  controls the capacitor's charging time, and so sets the debounce period for the condition when the switch opens. The equation for charging a capacitor is:

$$V_c = V_{final}(1 - e^{-t/RC})$$

Solving for  $R$ :  $R = -\frac{t}{C \ln\left(1 - \frac{V_{th}}{V_{final}}\right)}$  Here,  $R = R_1 + R_2$

$V_{final}$  is the final charged value of the 5 V power supply.  $V_{th}$  is the worst-case transition point for a high-going signal, which for our 74AHCT14 is 0.9 V, which is the lower threshold voltage,  $V_{th,L}$ . So,

$$R_1 + R_2 = \frac{20 \times 10^{-3}}{1 \times 10^{-6} \ln\left(1 - \frac{0.9}{5}\right)} = 100.78 \text{ k}\Omega \text{ (if no diode is used).}$$

Since, already we have calculated  $R_2 = 18 \text{ k}\Omega$ , hence,  $R_1 = 82 \text{ k}\Omega$ .

# Hardware Debouncing Example

The diode is an optional part needed. The use of diode forms a shortcut path that removes  $R_2$  from the charging circuit. All charges flow through only  $R_1$ . Equations will be the same except that we have to take the voltage drop across the diode into account. Change  $V_{final}$  to 4.3 V (5 V - 0.7 V, Si diode's forward voltage drop) and solve for  $R_1$  again, i.e.,

$$R_1 = \frac{20 \times 10^{-3}}{1 \times 10^{-6} \ln\left(1 - \frac{0.9}{4.3}\right)} = 85.16 \text{ k}\Omega \text{ (if a diode is used).}$$

Now,  $R_2 = 18 \text{ k}\Omega$  and  $R_1 = 82 \text{ k}\Omega$ , (available).

**Be wary of the components' tolerances!**

Standard resistors are usually  $\pm 5\text{-}10\%$  and the capacitors are  $\pm 20\%$  (for electrolytic) and  $\pm 30\%$  (for small ceramic capacitors) are error prone from their nominal rating.

# Software debouncing

- Debouncing in hardware may give **raise** to **additional cost**, and it is more difficult to determine a good debouncing for all the push button switches that will be used. So, it may be preferable to debounce the switch in software.
- While **numerous algorithms exist** to perform the debouncing function.



# Software debouncing

- One technique is to wait for a switch closure, then test the switch again after a short delay (e.g. 50 milliseconds or so). If it is still closed, we determine that the switch has changed state.
- Second technique is to test the switch periodically to see if it has changed state.

# Software Debouncing: Example Code-1

```
const int buttonPin = 2; // the pin that the button is connected to
int buttonState = 0;     // current state of the button
int lastButtonState = 0; // previous state of the button
unsigned long lastDebounceTime = 0; // the last time the button was pressed

void setup() {
  pinMode(buttonPin, INPUT);
  Serial.begin(9600); // initialize serial communication at 9600 bits per second
}

void loop() {
  // read the state of the button
  int reading = digitalRead(buttonPin);

  // check if the button state has changed
  if (reading != lastButtonState) {
    // reset the last debounce time
    lastDebounceTime = millis();
  }
```

```
// debounce the button
if ((millis() - lastDebounceTime) > 50) {
  // if the button state has changed, update the state variable
  if (reading != buttonState) {
    buttonState = reading;

    // print the new button state to the serial monitor
    Serial.print("Button state: ");
    Serial.println(buttonState);
  }
}

// save the current button state for the next iteration
lastButtonState = reading;
}
```

In this code, we use the **digitalRead** function to read the state of the button connected to pin 2. We then use a simple debounce algorithm that waits for a certain amount of time before updating the button state. This helps to eliminate any bouncing or false readings that may occur when the button is pressed.

The **lastDebounceTime** variable is used to store the last time the button was pressed, and the **millis** function is used to get the current time in milliseconds. If the time since the last button press is greater than 50 milliseconds, the button state is updated.

The current button state is stored in the **buttonState** variable, and the previous button state is stored in the **lastButtonState** variable. This allows us to check if the button state has changed since the last iteration of the **loop** function.

Finally, we print the new button state to the serial monitor using the **Serial.print** and **Serial.println** functions. This allows us to see the state of the button as we press it.

# Software Debouncing: Example Code-2

Each time the input pin goes from LOW to HIGH (e.g. because of a push-button press), the output pin is toggled from LOW to HIGH or HIGH to LOW. There's a minimum delay between toggles to debounce the circuit (i.e. to ignore noise or transition).

```
// constants won't change. They're used here to set pin numbers:  
const int buttonPin = 2; // the number of the pushbutton pin  
const int ledPin = 13;  // the number of the LED pin
```

```
// Variables will change:  
int ledState = HIGH;    // the current state of the output pin  
int buttonState;        // the current reading from the input pin  
int lastButtonState = LOW; // the previous reading from the input pin
```

```
// the following variables are unsigned longs because the time, measured in  
// milliseconds, will quickly become a bigger number than can be stored in an int.  
unsigned long lastDebounceTime = 0; // the last time the output pin was toggled  
unsigned long debounceDelay = 50;  // the debounce time; increase if the o/p flickers
```

## Software Debouncing: Example Code-2\_cont.

```
void setup() {  
  pinMode(buttonPin, INPUT);  
  pinMode(ledPin, OUTPUT);  
  
  // set initial LED state  
  digitalWrite(ledPin, ledState);  
}  
  
void loop() {  
  // read the state of the switch into a local variable:  
  int reading = digitalRead(buttonPin);  
  // check to see if you just pressed the button  
  // (i.e. the input went from LOW to HIGH), and you've waited long enough  
  // since the last press to ignore any noise:  
  
  // If the switch changed, due to noise or pressing:  
  if (reading != lastButtonState) {  
    // reset the debouncing timer  
    lastDebounceTime = millis();  
  }  
}
```

## Software Debouncing: Example Code-2\_cont.

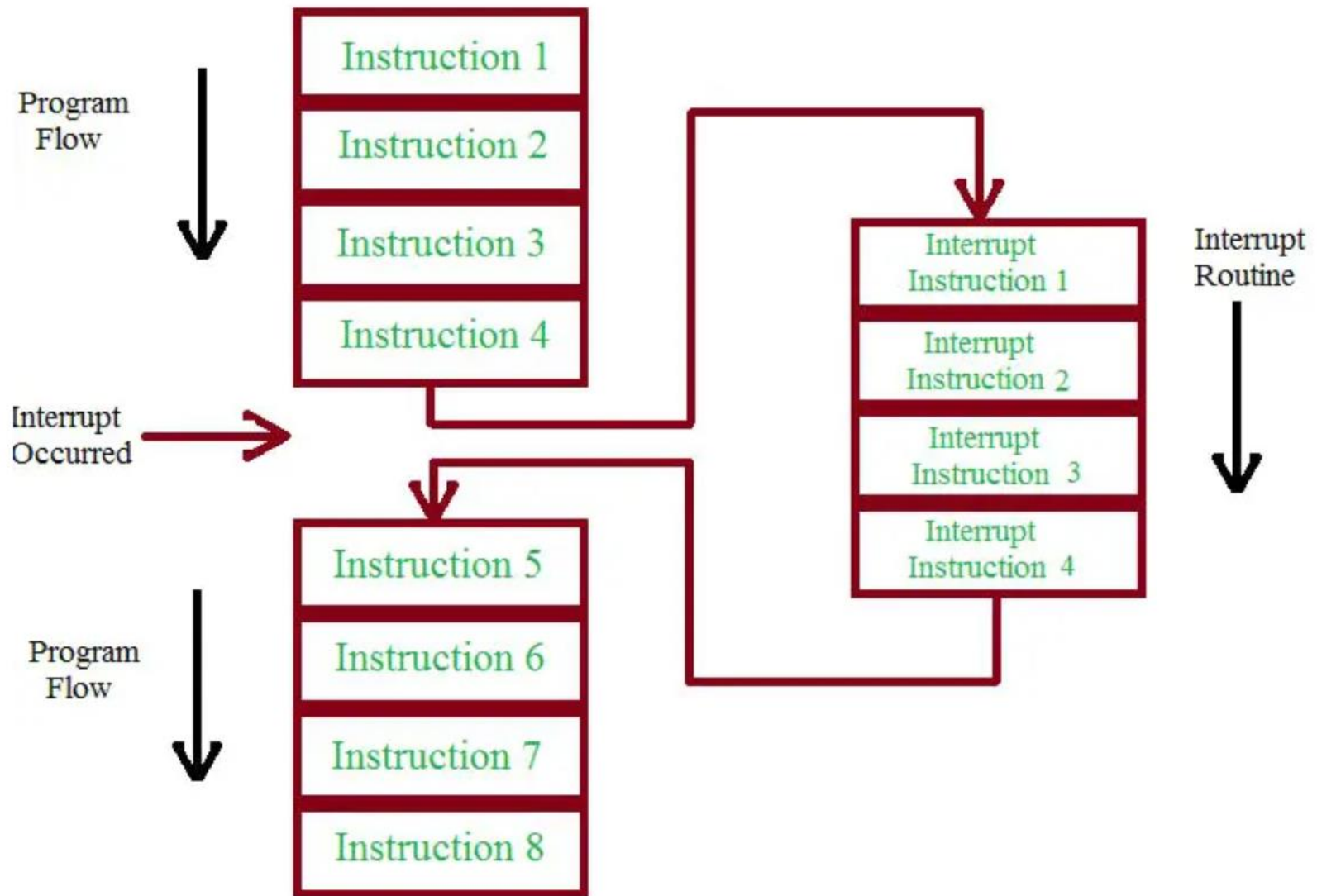
```
if ((millis() - lastDebounceTime) > debounceDelay) {  
  // whatever the reading is at, it's been there for longer than the debounce  
  // delay, so take it as the actual current state:  
  
  // if the button state has changed:  
  if (reading != buttonState) {  
    buttonState = reading;  
  
    // only toggle the LED if the new button state is HIGH  
    if (buttonState == HIGH) {  
      ledState = !ledState;  
    }  
  }  
}  
  
// set the LED:  
digitalWrite(ledPin, ledState);  
  
// save the reading. Next time through the loop, it'll be the lastButtonState:  
lastButtonState = reading;  
}
```

# Arduino Interrupts

# Interrupts

- A request for the processor to ‘interrupt’ or ‘suspend’ the **currently executing process** to execute **Interrupt Service Routine (ISR)**.
- Also referred to as ‘trap’
- If the request is accepted: the processor suspends current processes, saves its state, and executes **Interrupt Service Routine (ISR)** or, Interrupt Handler
- Concept of ‘Interrupt’ is very useful when implementing any of the switch debouncing methods.

# Interrupts...



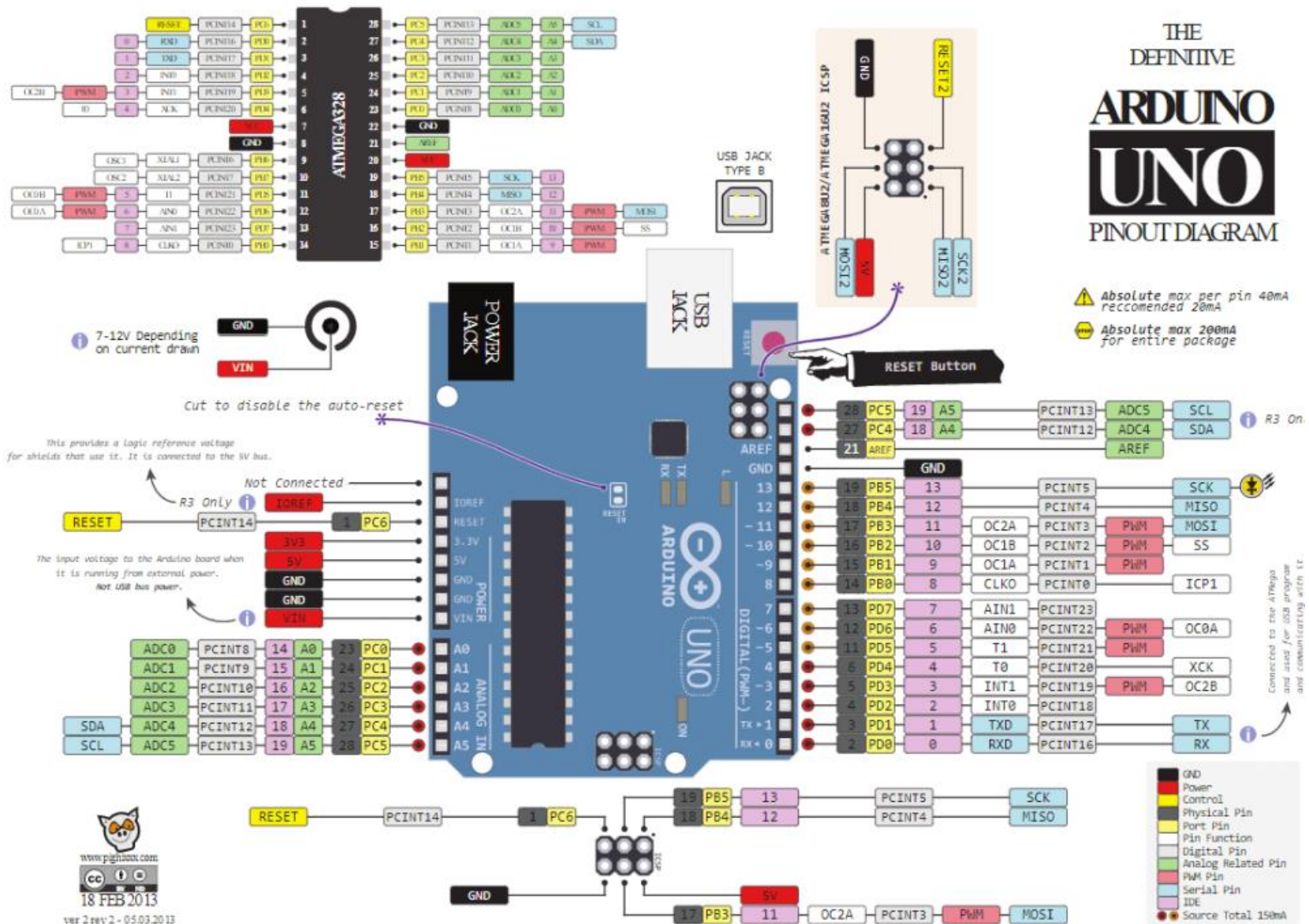


# Interrupts Categories

- Internal/Software Interrupts
- Hardware Interrupts
  - External Interrupts (triggered by **INT0** or **INT1** pins)
  - Pin Change (PC) Interrupts (triggered by Any of the **PCINT23:0** pins)

**N.B:** With the **INT** pins, you know that a change has occurred on that specific pin. With **PCINT**, the interrupt tells you that a pin-change interrupt has happened on some pin on the port associated with that interrupt. You then have to read a register to determine which pin it happened on

# Interrupts Categories



# Introduction

- Microcontroller normally executes instructions in an orderly fetch-execute sequence as dictated by a user-written program.
- However, a microcontroller must also be ready to handle **unscheduled**, events that might occur inside or outside the microcontroller.
- The interrupt system onboard a microcontroller allows it to respond to these internally and externally generated events. By definition, we do not know when these events will occur.
- When an interrupt event occurs, the microcontroller will normally complete the instruction it is currently executing and then transition program control to an Interrupt Service Routine (ISR). This ISR handles the interrupt.
- Once the ISR is complete, the microcontroller will resume processing where it left off before the interrupt event occurred.
- **Scheduled events are not interrupted**

# Why do we need it?

- **Let's think about a real-life example:**
- You are heating up your food using the microwave.
- In this case, you can either:
  - Stare at the microwave while it heats up the food
  - Go about your life normally and go to the microwave when you get the signal that the food has been heated up
- **Which of the above-mentioned options do you think is more efficient?**
- Usually, we want to go about our life while the food is being heated up.
- This is because we want to make efficient use of our time.
- We hear the ting sound from the microwave, stop what we are doing, and go to take out the heated food.
- This 'ting' sound is an interrupt.
- It is very similar for processors as well.
- **Similarly, if the processor is waiting for some conditions to be fulfilled to perform a specific task (task 1):**
  - It can wait and halt other processes till the conditions for the main routine is fulfilled.
  - It can execute the normal instructions and halt them for a brief time once the conditions for task 1 have been fulfilled, finish task 1 and go back to executing the main routine.
- **Clearly, the 2<sup>nd</sup> option is better for maximizing the processor's resource utilization.**
- This is where interrupts come in handy.
- When the conditions for task 1 has been met, it can simply **trigger an interrupt**, stop the main routine to complete the task-1 and once finished go back to the main routine.

# The Main Reasons You Might Use Interrupts

- To detect pin changes (eg. rotary encoders, button presses)
- Watchdog timer (eg. if nothing happens after 8 seconds, interrupt me)
- **Timer interrupts - used for comparing/overflowing timers**
- SPI data transfers
- I2C data transfers
- USART data transfers
- ADC conversions (analog to digital)
- EEPROM ready for use
- Flash memory ready

# Interrupt Vector

- The interrupt vectors and vector table are crucial to the understanding of hardware and software interrupts. **Interrupt vectors are addresses** that inform the interrupt handler as to where to find the ISR (Interrupt Service Routine, also called Interrupt Service Procedure).
- Misspelling the vector name (even wrong capitalization) will result in the ISR not being called and **it will not also result a compiler error.**

# Interrupt Function of Arduino


- Re-enables interrupts (after they've been disabled by `noInterrupts()`). Interrupts allow certain important tasks to happen in the background and are enabled by default. Some functions will not work while interrupts are disabled, and incoming communication may be ignored. Interrupts can slightly disrupt the timing of code, however, and may be disabled for particularly critical sections of code.
  - Syntax: `noInterrupts()` and `interrupts()` for Arduino IDE only  
`cli()` and `sei()` for both Arduino IDE and ATmega328P
  - Parameters: None
  - Returns: Nothing
- **Example Code:** The code enables Interrupts.

```
void setup() {}  
void loop() {  
  noInterrupts();  
  // critical, time-sensitive code here  
  interrupts();  
  // other code here  
}
```

# Global Interrupt Enable

- The main interrupt flag
- This is used to turn **all the interrupts on or off**
- Example: `sei();` //globally enable interrupt
- Available in the Status Register (SREG): Bit 7

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	





## Atmega328p Interrupt Vector Table

- The ATmega328P provides support for **25 different interrupt sources**. These interrupts and the separate Reset Vector each have a separate program vector located at the lowest addresses in the Flash program memory space.
- The complete list of “Reset and Interrupt Vectors” in ATmega328P is shown in this table. Each Interrupt Vector occupies two instruction words.
- The list also determines the priority levels of the different interrupts. **The lower the address the higher is the priority level.** RESET has the highest priority, and next is INT0 – the External Interrupt Request 0.

# Atmega328p Interrupt Vector Table

VectorNo.	Program Address <sup>(2)</sup>	Source	Interrupt Definition
1	0x0000 <sup>(1)</sup>	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/Counter1 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

**Highest  
Priority**

**Lowest  
Priority**

# Trigger

- **An asynchronous event** which causes the interrupt
- For example, the push button press during experiment 4 in our MES Lab can be a trigger.
- It can cause an interrupt which is registered by the module and is reacted to.
- However, what if all the conditions are not met but a trigger flag is set?
- In this case, rather than the request being dismissed, it is held pending, postponed until a later time.

# What happens after a trigger?

- Once the **interrupt is triggered and processed**, the interrupt flag is cleared.
- **Clearing the interrupt flag** is called **acknowledgement**.

# Interrupt Service Routine (ISR)

- The module that is executed when hardware requests an interrupt.
- There may be 1 large ISR handling all the interrupt requests, or many small ISRs handling the many interrupts (interrupt vectors).

## Example:

ISR (TIMER0\_OVF\_vect) //enabling overflow vector inside Timer0 using an ISR

ISR (TIMER0\_COMPA\_vect) //This is the Timer0 Compare 'A' interrupt service routine.

**Remember, the ISR is a separate routine and requires a separate flowchart to represent.**



## TIMSK0 - Timer/Counter0 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7..3 – Res: Reserved Bits**

These bits are reserved bits in the ATmega328P and will always read as zero.

- **Bit 2 – OCIE0B: Timer/Counter Output Compare Match B Interrupt Enable**

When the **OCIE0B** bit of **TIMSK0** register is written to ‘one’, and the **I**-bit in the Status Register - **SREG** is ‘set’, the Timer/Counter Compare Match B interrupt is **enabled**. The corresponding interrupt is **executed** if **compare match** in Timer/Counter0 occurs, i.e., when the **OCF0B** bit is ‘set’ in the Timer/Counter Interrupt Flag Register – **TIFR0**.

# TIMSK0 - Timer/Counter0 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 1 – OCIE0A: Timer/Counter0 Output Compare Match A: Interrupt Enable**

When the OCIE0A bit is written to one, and the I-bit in the Status Register is set, the Timer/Counter0 Compare Match A interrupt is enabled. The corresponding interrupt is executed if a Compare Match in Timer/Counter0 occurs, i.e., when the OCF0A bit is set in the Timer/Counter 0 Interrupt Flag Register – TIFR0.

- **Bit 0 – TOIE0: Timer/Counter0 Overflow Interrupt Enable**

When the **TOIE0** bit of **TIMSK0** register is written to ‘one’, and the **I**-bit in the Status Register - **SREG** is ‘set’, the Timer/Counter0 Overflow interrupt is **enabled**. The corresponding interrupt is **executed** if an **overflow** in Timer/Counter0 **occurs**, i.e., when the **TOV0** bit is ‘set’ in the Timer/Counter0 Interrupt Flag Register – **TIFR0**



## TIFR0 - Timer/Counter0 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x15 (0x35)	–	–	–	–	–	OCF0B	OCF0A	TOV0	TIFR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7..3 – Res: Reserved Bits**

These bits are reserved bits in the ATmega328P and will always read as zero.

- **Bit 2 – OCF0B: Timer/Counter 0 Output Compare B Match Flag**

The OCF0B bit is set when a Compare Match occurs between the Timer/Counter and the data in OCR0B – Output Compare Register0 B. OCF0B is cleared by hardware when executing the corresponding interrupt handling vector.

Alternatively, OCF0B is cleared by writing a logic one to the flag. When the **I**-bit in **SREG**, **OCIE0B** (Timer/Counter Compare B Match Interrupt Enable) bit in **TIMSK0** register, and **OCF0B** bit in **TIFR0** register are ‘set’, the Timer/Counter Compare Match Interrupt is **executed**.

## TIFR0 - Timer/Counter0 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x15 (0x35)	–	–	–	–	–	OCF0B	OCF0A	TOV0	TIFR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 1 – OCF0A: Timer/Counter 0 Output Compare A Match Flag**

The OCF0A bit is set when a Compare Match occurs between the Timer/Counter0 and the data in OCR0A – Output Compare Register0. OCF0A is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, OCF0A is cleared by writing a logic one to the flag. When the I-bit in SREG, OCIE0A (Timer/Counter0 Compare Match Interrupt Enable), and OCF0A are set, the Timer/Counter0 Compare Match Interrupt is executed.

## TIFR0 - Timer/Counter0 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x15 (0x35)	-	-	-	-	-	OCF0B	OCF0A	TOV0	TIFR0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 0 – TOV0: Timer/Counter0 Overflow Flag**

The bit TOV0 in TIFR0 register is set when an overflow occurs in Timer/Counter0. TOV0 is cleared by hardware when executing the corresponding interrupt handling vector.

Alternatively, TOV0 bit in TIFR0 register is cleared by writing a logic one to the flag. When the **I**-bit in **SREG**, **TOIE0** (Timer/Counter0 Overflow Interrupt Enable) bit in **TIMSK0** register and **TOV0** bit in **TIFR0** register are ‘set’, the Timer/Counter0 Overflow interrupt is **executed**.

The setting of this flag is dependent on the WGM02:0 bit setting

## TIMSK1 - Timer/Counter1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6F)	–	–	ICIE1	–	–	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7, 6 – Res: Reserved Bits**

These bits are unused bits in the ATmega48P/88P/168P/328P and will always read as zero.

- **Bit 5 – ICIE1: Timer/Counter1, Input Capture Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Input Capture interrupt is enabled. The corresponding Interrupt Vector is executed when the ICF1 Flag, located in TIFR1, is set.

## TIMSK1 - Timer/Counter1 Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6F)	-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1	TIMSK1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 4, 3 – Res: Reserved Bits**

These bits are unused bits in the ATmega328P and will always read as 0.

- **Bit 2 – OCIE1B: Timer/Counter1, Output Compare B Match Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Output Compare B Match interrupt is enabled. The corresponding Interrupt Vector is executed when the OCF1B Flag, located in TIFR1, is set.

## TIMSK1 - Timer/Counter1 Interrupt Mask Register

- **Bit 1 – OCIE1A: Timer/Counter1, Output Compare A Match Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Output Compare A Match interrupt is enabled. The corresponding Interrupt Vector is executed when the OCF1A Flag, located in TIFR1, is set.

- **Bit 0 – TOIE1: Timer/Counter1, Overflow Interrupt Enable**

When this bit is written to one, and the I-flag in the Status Register is set (interrupts globally enabled), the Timer/Counter1 Overflow interrupt is enabled. The corresponding Interrupt Vector is executed when the TOV1 Flag, located in TIFR1, is set.

## TIFR1 - Timer/Counter1 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x16 (0x36)	–	–	<b>ICF1</b>	–	–	<b>OCF1B</b>	<b>OCF1A</b>	<b>TOV1</b>	<b>TIFR1</b>
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7, 6 – Res: Reserved Bits**

These bits are unused bits in the ATmega328P and will always read as zero.

- **Bit 5 – ICF1: Timer/Counter1, Input Capture Flag**

This flag is set when a capture event occurs on the ICP1 pin. When the Input Capture Register (ICR1) is set by the WGM13:0 to be used as the TOP value, the ICF1 Flag is set when the counter reaches the TOP value. ICF1 is automatically cleared when the Input Capture Interrupt Vector is executed. Alternatively, ICF1 can be cleared by writing a logic one to its bit location.

## TIFR1 - Timer/Counter1 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x16 (0x36)	–	–	<b>ICF1</b>	–	–	<b>OCF1B</b>	<b>OCF1A</b>	<b>TOV1</b>	<b>TIFR1</b>
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 4, 3 – Res: Reserved Bits**

These bits are unused bits in the ATmega328P, and will always read as zero.

- **Bit 2 – OCF1B: Timer/Counter1, Output Compare B Match Flag**

This flag is set in the timer clock cycle after the counter (TCNT1) value matches the Output Compare Register B (OCR1B). Note that a Forced Output Compare (FOC1B) strobe will not set the OCF1B Flag. OCF1B is automatically cleared when the Output Compare Match B Interrupt Vector is executed. Alternatively, OCF1B can be cleared by writing a logic one to its bit location.



## TIFR1 - Timer/Counter1 Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x16 (0x36)	–	–	ICF1	–	–	OCF1B	OCF1A	TOV1	TIFR1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 1 – OCF1A: Timer/Counter1, Output Compare A Match Flag**

This flag is set in the timer clock cycle after the counter (TCNT1) value matches the Output Compare Register A (OCR1A). Note that a Forced Output Compare (FOC1A) strobe will not set the OCF1A Flag. OCF1A is automatically cleared when the Output Compare Match A Interrupt Vector is executed. Alternatively, OCF1A can be cleared by writing a logic one to its bit location.

- **Bit 0 – TOV1: Timer/Counter1, Overflow Flag**

The setting of this flag is dependent of the WGM13:0 bits setting. In Normal and CTC modes, the TOV1 Flag is set when the timer overflows.

Flag behavior when using another WGM13:0 bit setting. TOV1 is automatically cleared when the Timer/Counter1 Overflow Interrupt Vector is executed. Alternatively, TOV1 can be cleared by writing a logic one to its bit location.

## 500 ms Blink Example Code using Timer1 Interrupts

Timer 1 is used here. The values of TCCR1A and 1B are reset to 0 to make sure everything is clear. TCCR1B was set equal to 00000100 for a prescaler of 256. If you want to set ones, an OR operation can be used. If you want to set zeros, an AND operation can be used. The compare match mode for the OCR1A register is enabled. For that, the OCIE1A bit was set to be a 1 and that's from the TIMSK1 register. So, we equal that to OR and this byte 00000010.

The OCR1A register was set to 31250 value so that we will have an interruption each 500ms. Each time the interrupt is triggered, we go to the related ISR vector. Since we have **3 timers**, we have **6 ISR vectors**, two for each timer and they have these names:

TIMER1\_COMPA\_vect,    TIMER2\_COMPA\_vect,    TIMERO\_COMPA\_vect,  
 TIMER1\_COMPB\_vect,    TIMER2\_COMPB\_vect,    TIMERO\_COMPB\_vect

## 500 ms Blink Example Code using Timer1 Interrupts

**Timer 1** and compare register A was used so we need to use the ISR **TIMER1\_COMPA\_vect**. So below the void loop, the interruption routine is defined. Inside this interruption, the state of the LED was inverted, and a digital write was created. But first, **the timer value** was reset. Otherwise, it will continue to count up to its maximum value. So, each 500ms, this code will run and invert the LED state and that creates a blink of the LED connected to pin D5 for example.

## 500 ms Blink Example Code using Timer1 Interrupts and Pre-scalar values of 256

```

Calculations (for 500ms):
System clock 16 Mhz and Prescalar 256;
Timer 1 speed = 16Mhz/256 = 62.5 Khz
Pulse time = 1/62.5 Khz = 16us
Count up to = 500ms / 16us = 31250 (so this is the value the OCR register should have)*/
bool LED_STATE = true;

void setup() {
  pinMode(13, OUTPUT);      //Set the pin to be OUTPUT
  cli();                    //stop interrupts for till we make the settings
  /*1. First we reset the control register to amke sure we start with everything disabled.*/
  TCCR1A = 0;               // Reset entire TCCR1A to 0
  TCCR1B = 0;               // Reset entire TCCR1B to 0

  /*2. We set the prescalar to the desired value by changing the CS10 CS12 and CS12 bits. */
  TCCR1B |= B00000100;      //Set CS12 to 1 so we get prescalar 256

  /*3. We enable compare match mode on register A*/
  TIMSK1 |= B00000010;      //Set OCIE1A to 1 so we enable compare match A

  /*4. Set the value of register A to 31250*/
  OCR1A = 31250;            //Finally we set compare register A to this value
  sei();                    //Enable back the interrupts
}

```

CSx2	CSx1	CSx0	Prescaler
0	0	1	1
0	1	0	8
0	1	1	64
1	0	0	256
1	0	1	1024

Bit	7	6	5	4	3	2	1	0
(0x6F)	-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	1	0

OCR1A register value is compared with TCNT1 register value

## 500 ms Blink Example Code using Timer1 Interrupts

```
void loop() {  
    // put your main code here, to run repeatedly:  
}  
  
//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(13,LED_STATE); //Write new state to the LED on pin D5  
}
```

## 500 ms Blink Example Code using Timer1 Interrupts

```
bool LED_State = 'True';

void setup() {
  pinMode(13, OUTPUT);
  cli();    // stop interrupts till we make the settings
  TCCR1A = 0; // Reset the entire A and B registers of Timer1 to make sure that
  TCCR1B = 0; // we start with everything disabled.
  TCCR1B = 0b00000100; // Set CS12 bit of TCCR1B to 1 to get a prescalar value of 256.
  TIMSK1 = 0b00000010; // Set OCIE1A bit to 1 to enable compare match mode of A reg.
  OCR1A = 31250; // We set the required timer count value in the compare register, A
  sei();    // Enable back the interrupts
}

void loop() {
  // put your main code here, to run repeatedly.
}

// With the settings above, this ISR will trigger each 500 ms.
ISR(TIMER1_COMPA_vect){
  TCNT1 = 0;    // First, set the timer back to 0 so that it resets for the next interrupt
  LED_State = !LED_State; // Invert the LED State
  digitalWrite(13, LED_State); // Write this new state to the LED connected to pin D5
}
```

# 500 ms Blink Example Code using Timer1 Interrupts

**//If you increase prescaler value to 101 then delay would rise.**

```
bool LED_State = 'True';
```

```
void setup() {
  pinMode(13, OUTPUT);
  cli();    // stop interrupts till we make the settings
  TCCR1A = 0; // Reset the entire A and B registers of Timer1 to make sure that
  TCCR1B = 0; // we start with everything disabled.
  TCCR1B = 0b00000101; // Set CS12 bit of TCCR1B to 1 to get a prescaler value of 256.
  TIMSK1 = 0b00000010; // Set OCIE1A bit to 1 to enable compare match mode of A reg.
  OCR1A = 31250; // We set the required timer count value in the compare register, A
  sei();    // Enable back the interrupts
}
```

```
void loop() {
  // put your main code here, to run repeatedly.
}
```

// With the settings above, this ISR will trigger each 500 ms.

```
ISR(TIMER1_COMPA_vect){
  TCNT1 = 0;    // First, set the timer back to 0 so that it resets for the next interrupt
  LED_State = !LED_State; // Invert the LED State
  digitalWrite(13, LED_State); // Write this new state to the LED connected to pin D5
}
```

# Hardware Interrupts

- Hardware interrupts triggered by:
  - `INT0` or `INT1` pins
  - Any of the `PCINT23:0` pins
- `INT0` and `INT1` are mapped with the pins `PD2` and `PD3` on ATmega328P
- `PD2` and `PD3` are mapped with digital pins 2 and 3 (GPIO) on the Arduino Uno board
- `PCINT23:0` are mapped with ports B, C, and D on the Arduino Uno board
- Implies any of the I/O pins on the board can be configured to handle interrupts
- These interrupts can be used to wake up the processor from sleep modes



# Hardware Interrupts

- INTO and INT1 are referred to as 'External Interrupts'
  - Can be triggered by a rising edge, falling edge, logical change, or low level
- PCINT23:0 are referred to as 'Pin Change Interrupts'
- External interrupts have a higher priority
- External interrupts have their own interrupt vectors
- Pin change interrupts share interrupt vectors
  - The interrupt handler has to decide which pin the interrupt originated from

## EICRA - External Interrupt Control Register A

- Contains control bits for interrupt sense control

Bit	7	6	5	4	3	2	1	0	
(0x69)	-	-	-	-	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bits 7:4 – Reserved bits – always read as 0
- Bits 3:2 – ISC11, ISC10 - Interrupt Sense Control 1
- Bits 1:0 – ISC01, ISC00 – Interrupt Sense Control 0
- ISC1x corresponds to INT1
- ISC0x corresponds to INT0

## EICRA - External Interrupt Control Register A

- External Interrupts activated by the external pin INT1 or INT0 if the SREG I-flag and the corresponding interrupt mask are set.
- Level/edge on INT1/INT0 pin which activates interrupts:

ISCx1	ISCx0	Description
0	0	Low level of INT1/INT0 generates interrupt request
0	1	Any logical/level change on INT1/INT0 generates interrupt request
1	0	Falling edge of INT1/INT0 generates interrupt request
1	1	Rising edge of INT1/INT0 generates interrupt request

## EICRA - External Interrupt Control Register A

- Value of the pin is sampled before detecting the edges
- If edge or toggle interrupt is selected:
  - Pulses longer than 1 clock period will generate an interrupt
  - Shorter pulses are not guaranteed to generate an interrupt
- If low-level interrupt is selected, to generate interrupt:
  - Low level must be held until the completion of currently executing instruction

## EIMSK – External Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
0x1D (0x3D)	-	-	-	-	-	-	INT1	INT0	EIMSK
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bits 7:2 – Reserved – always read as 0
- Bit 1 – INT1: External Interrupt Request 1 Enable
- Bit 0 – INT0: External Interrupt Request 0 Enable

## EIMSK - External Interrupt Mask Register

- When INT1/INT0 bit is enabled, and I bit (bit 7) of SREG is enabled:
  - External pin interrupt is activated
  - ISCx1, and ISCx0 pins in EICRA define whether the external interrupt will be activated on:
    - Rising edge
    - Falling edge
    - Logic change
    - Low level
- Activity on the pin will cause an interrupt even if the pin is configured as an output
- Executed from INT1/INT0 interrupt vector

## EIFR – External Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x1C (0x3C)	-	-	-	-	-	-	INTF1	INTF0	EIFR
Read/Write	R	R	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bits 7:2 – Reserved – always read as 0
- Bit 1 – INTF1: External Interrupt Flag 1
- Bit 0 – INTF0: External Interrupt Flag 0

## EIFR - External Interrupt Flag Register

- When the INTx pin triggers an interrupt request, INTFx is set
- If I bit in SREG and INTx bit in EIMSK are set, MCU will jump to a relevant interrupt vector
- INTFx cleared when:
  - Interrupt routine is executed
  - A logical 1 is written (cancel any falling interrupts)
  - Always cleared when INTx is configured as a level interrupt



## PCICR - Pin Change Interrupt Control Register

Bit	7	6	5	4	3	2	1	0	
(0x68)	-	-	-	-	-	PCIE2	PCIE1	PCIE0	PCICR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bits 7: 3 – Reserved – always read as 0
- Bit 2 – PCIE2 – Pin Change Interrupt Enable 2
- Bit 1 – PCIE1 – Pin Change Interrupt Enable 1
- Bit 0 – PCIE0 – Pin Change Interrupt Enable 0

## PCICR - Pin Change Interrupt Control Register

- If PCIE<sub>x</sub> is enabled and I bit in SREG is enabled, pin change interrupt x is enabled
- PCINT 23:16 corresponds to pin change interrupt 2
- PCINT 14:8 corresponds to pin change interrupt 1
- PCINT 7:0 corresponds to pin change interrupt 0
- The pins are enabled individually from PCMSK<sub>x</sub> register
- Executed from PCIF<sub>x</sub> interrupt vector
- Any change in these pins will cause an interrupt

# PCICR - Pin Change Interrupt Control Register

- PCINT 23:16 –
  - PD7:PDo on ATMega328P
  - Pins D7:Do (GPIO) on Arduino Uno board
- PCINT 14:8 –
  - PC6:PCo on ATMega328P
  - PC5:0 – Pins A5:A0 on Arduino Uno board
  - PC6 – Reset pin on Arduino Uno Board
- PCINT 7:0 –
  - PB7:PBo on ATMega328P
  - PB5:0 – Pins D13:D8 on Arduino Uno board
  - PB6 and PB7 are not available externally on the Arduino Uno board

## PCIFR - Pin Change Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x1B (0x3B)	-	-	-	-	-	PCIF2	PCIF1	PCIF0	PCIFR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bits 7:3 – Reserved – Always read as 0
- Bit 2 – PCIF2 – Pin Change Interrupt Flag 2
- Bit 1 – PCIF1 – Pin Change Interrupt Flag 1
- Bit 0 – PCIF0 – Pin Change Interrupt Flag 0

## PCIFR - Pin Change Interrupt Flag Register

- Any logic change on PCINT23:0 triggers an interrupt request: PCIFx set
- If I bit in SREG and PCIE<sub>x</sub> on PCICR are set, MCU will go to the corresponding interrupt vector
- Flag is cleared when:
  - Interrupt routine is executed
  - A logical 1 is written

## PCMSK2 - Pin Change Mask Register 2

Bit	7	6	5	4	3	2	1	0	
(0x6D)	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16	PCMSK2
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bits 7:0 – PCINT23:16 – Pin Change Enable Mask 23:16
- Each bit selects whether the pin change interrupt is enabled on the corresponding I/O pin
- If any bit from PCINT23:16 is set and the PCIE2 bit on PCICR is set, pin change interrupt is enabled on the corresponding I/O pin
- If PCINT23:16 is cleared, the pin change interrupt on the corresponding I/O pin is disabled

## PCMSK1 - Pin Change Mask Register 1

Bit	7	6	5	4	3	2	1	0	
(0x6C)	–	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	PCMSK1
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7 – Reserved – Always read as 0
- Bits 6:0 – PCINT14:8 – Pin Change Enable Mask 14:8
- Each bit selects whether the pin change interrupt is enabled on the corresponding I/O pin
- If any bit from PCINT14:8 is set and the PCIE1 bit on PCICR is set, pin change interrupt is enabled on the corresponding I/O pin
- If PCINT14:8 is cleared, the pin change interrupt on the corresponding I/O pin is disabled

# PCMSK0 – Pin Change Mask Register 0

Bit	7	6	5	4	3	2	1	0	
(0x6B)	<b>PCINT7</b>	<b>PCINT6</b>	<b>PCINT5</b>	<b>PCINT4</b>	<b>PCINT3</b>	<b>PCINT2</b>	<b>PCINT1</b>	<b>PCINT0</b>	PCMSK0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bits 7:0 – PCINT7:0 – Pin Change Enable Mask 7:0
- Each bit selects whether the pin change interrupt is enabled on the corresponding I/O pin
- If any bit from PCINT7:0 is set and the PCIE0 bit on PCICR is set, pin change interrupt is enabled on the corresponding I/O pin
- If PCINT7:0 is cleared, the pin change interrupt on the corresponding I/O pin is disabled



# References

- ATMega328 Datasheet
- Arduino Uno Datasheet
- Chapter 5: Microprocessors, Advanced Industrial Control Technology by Peng Zhang
- <http://www.ganssle.com/debouncing-pt2.htm>
- <https://mansfield-devine.com/speculatrix/2018/04/debouncing-fun-with-schmitt-triggers-and-capacitors/>
- <https://www.arxterra.com/10-atmega328p-interrupts/>
- <https://www.arxterra.com/11-atmega328p-external-interrupts/>
- <http://www.gammon.com.au/forum/?id=11488>
- <http://www.ganssle.com/debouncing-pt2.htm>
- <https://mansfield-devine.com/speculatrix/2018/04/debouncing-fun-with-schmitt-triggers-and-capacitors/>
- [http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C12\\_Interrupts.htm](http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C12_Interrupts.htm)
- <https://docs.arduino.cc/built-in-examples/digital/Debounce>

## Appendix: Code -- Debounce

```
// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 2; // the number of the pushbutton pin
const int ledPin = 13;   // the number of the LED pin

// Variables will change:
int ledState = HIGH;      // the current state of the output pin
int buttonState;          // the current reading from the input pin
int lastButtonState = LOW; // the previous reading from the input pin

// the following variables are unsigned long's because the time, measured in milliseconds,
// will quickly become a bigger number than can be stored in an int.
unsigned long lastDebounceTime = 0; // the last time the output pin was toggled
unsigned long debounceDelay = 50;   // the debounce time; increase if the output flickers

void setup() {
  pinMode(buttonPin, INPUT);
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
  // set initial LED state
  digitalWrite(ledPin, ledState);
}
```

## Appendix: Code -- Debounce

```
void loop() {  
  // read the state of the switch into a local variable:  
  int reading = digitalRead(buttonPin);  
  
  // check to see if you just pressed the button  
  // (i.e. the input went from LOW to HIGH), and you've waited  
  // long enough since the last press to ignore any noise:  
  
  // If the switch changed, due to noise or pressing:  
  if (reading != lastButtonState) {  
    // reset the debouncing timer  
    lastDebounceTime = millis();  
  }  
}
```

## Appendix: Code -- Debounce

```
if ((millis() - lastDebounceTime) > debounceDelay) {  
  // whatever the reading is at, it's been there for longer  
  // than the debounce delay, so take it as the actual current state:  
  
  // if the button state has changed:  
  if (reading != buttonState) {  
    buttonState = reading;  
  
    // only toggle the LED if the new button state is HIGH  
    if (buttonState == HIGH) {  
      ledState = !ledState;  
    }  
    Serial.print("Button state: ");  
    Serial.println(buttonState);  
  
  }  
}  
  
// set the LED:  
digitalWrite(ledPin, ledState);  
  
// save the reading. Next time through the loop,  
// it'll be the lastButtonState:  
lastButtonState = reading;  
}
```

# Appendix: Code -- Interrupt

```

bool LED_state = 'True';
//timer delay
int delay_timer(int milliseconds)
{
    int count = 0;
    while(1)
    {
        if(TCNT2 >= 16) // Checking if 1 millisecond has passed
        {
            TCNT2=0;
            count++;
            if (count == milliseconds) {
                count=0;
                break; // exits the loop
            }
        }
    }
    return 0;
}

```

```

void setup(){
    Serial.begin(9600);
    //TIMER0 has been used for user defined delay;
    "delay_timer();"
    TCCR0A=0x00;
    TCCR0B=0x05;
    TCNT0=0;

    //TIMER1 has been used for OCR1A
    TCCR1A=0;
    TCCR1B=0;
    TCCR1B=0b000000100;
    TIMSK1=0b000000010;
    OCR1A=31250;
}

```

## Appendix: Code -- Interrupt

```
void loop(){

  Serial.println("Main Routine");
  delay_timer(4000);
  sei(); //enable interrupt
  //cli();//clear interrupt
}

ISR(TIMER1_COMPA_vect){

  TCNT1=0;
  LED_state=!LED_state;
  digitalWrite(13, LED_state);
  Serial.println("Inside ISR Routine");
  delay_timer(4000);
}
```

Thanks for attending....

