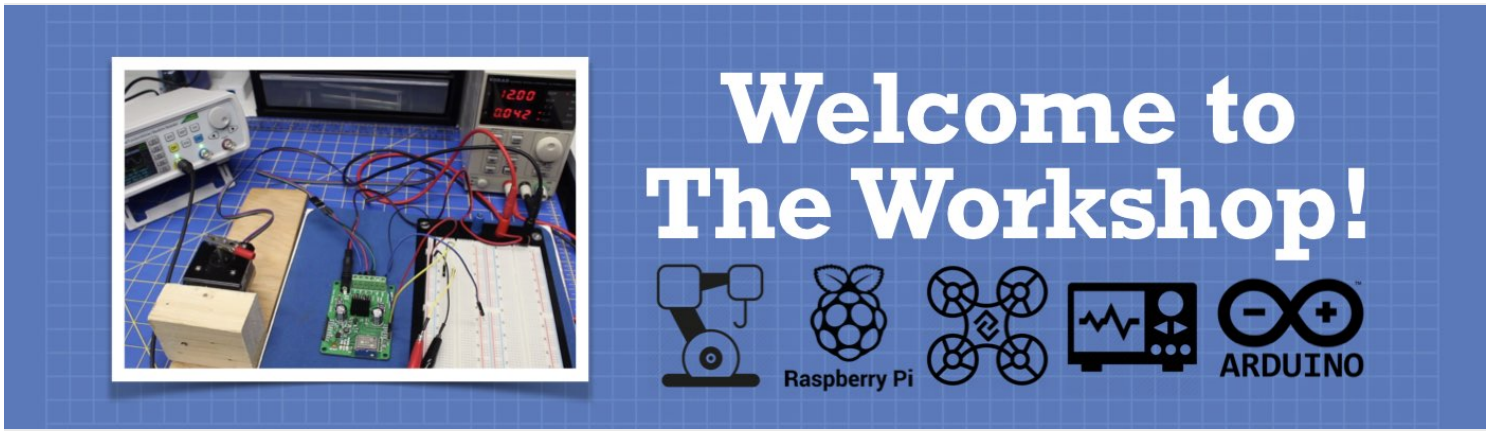


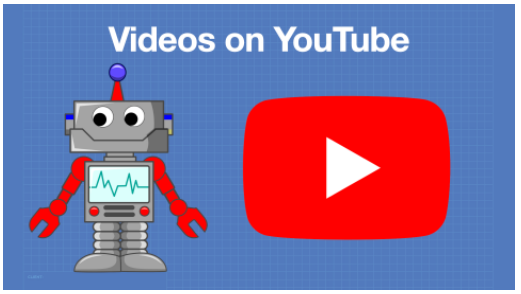
DroneBot Workshop

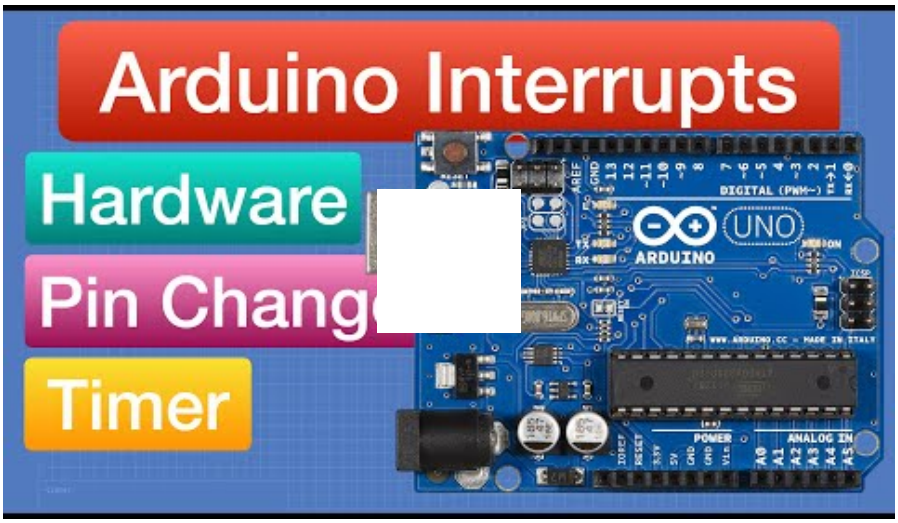
Arduino, Electronics, IoT, Raspberry Pi and Robots – Welcome to the Workshop!

Using Arduino Interrupts – Hardware, Pin Change and Timer

Table of Contents [show]

Today we will be learning about interrupts, a very important and fundamental feature of the Arduino and other microcontrollers. While we will be focusing on an Arduino Uno, the concepts presented here are equally valid with other boards.





Let's keep in touch!

Please subscribe to the newsletter and keep up to date with what is happening in the workshop.

Zero spam, no sales - just useful information!

Subscribe Today!

Introduction

When we design a project, we usually base it upon a microcontroller. There are a lot of good reasons for doing that, among them:

- Microcontrollers can handle multiple inputs and outputs.
- Microcontrollers can provide precision timing pulses.
- Microcontrollers are fast.

Because they can handle multiple inputs, and because they can do many things, microcontrollers can get quite busy. And busy microcontrollers need a way of managing external events, like pushbutton presses, while juggling other inputs and outputs timing processes.

One way of keeping control of external inputs, or internal timing events, is to use interrupts.

Latest Articles

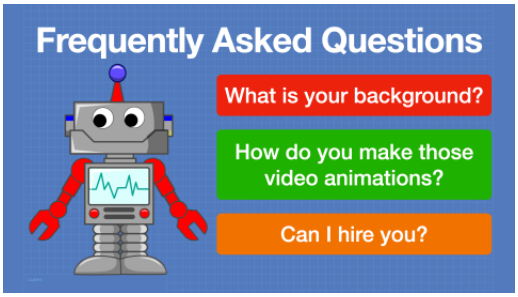


Table of Contents

How Interrupts Work

Interrupts are pretty well what the name implies, a method of interrupting the execution of a program in order to take care of something else.

Interrupts are by no means unique to microcontrollers, they have been used in computers and controllers for decades. When you type on a keyboard, move a mouse, or swipe on a touchscreen, you are creating interrupts, and these interrupts get services working that create the appropriate response to your actions.

In its basic form, an interrupt works like this:

- A program is running.
- An interrupt occurs.
- The program is paused, and its data is put aside so that it can resume later.
- The code related to the interrupt is run.
- When the interrupt code has finished, the program resumes where it left off.

Interrupts are great for monitoring events such as switch presses or alarm triggers, which occur spasmodically. They are also the proper choice when you need to measure input pulses accurately.

There are many types of interrupts used with microcontrollers and microprocessors, the interrupt features vary from model to model. They can all be divided broadly into two categories:

- **Hardware Interrupts –** These usually come from external signals.
- **Software Interrupts** – These are internal signals, usually controlled by timers or by software-related events.

Using interrupts will make you a better coder, and they are not that hard to use once you get familiar with them. Today, we will see how to use interrupts with an Arduino Uno.

Arduino Uno Interrupts

The Arduino Uno supports three types of interrupts:

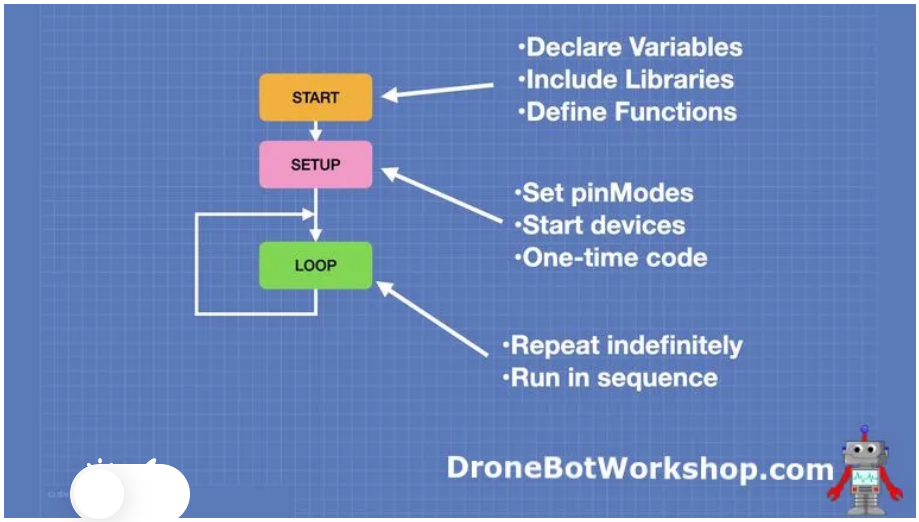
- **Hardware Interrupts** – External interrupt signals on specific pins.
- **Pin Change Interrupts** – External interrupts on any pin, grouped into ports.
- **Timer Interrupts** – Internal timer-generated interrupts, manipulated in software.

We will discuss these in detail in a bit, but for now, let's just say that they all work in basically the same way. When an interrupt event occurs, the microcontroller runs some code that you have placed in an "Interrupt Service Routine" or ISR function.

Where Interrupts fit in

Let's look at how all of this fits into your Arduino sketch.

We can visualize an Arduino sketch with a simple flowchart, something like this:

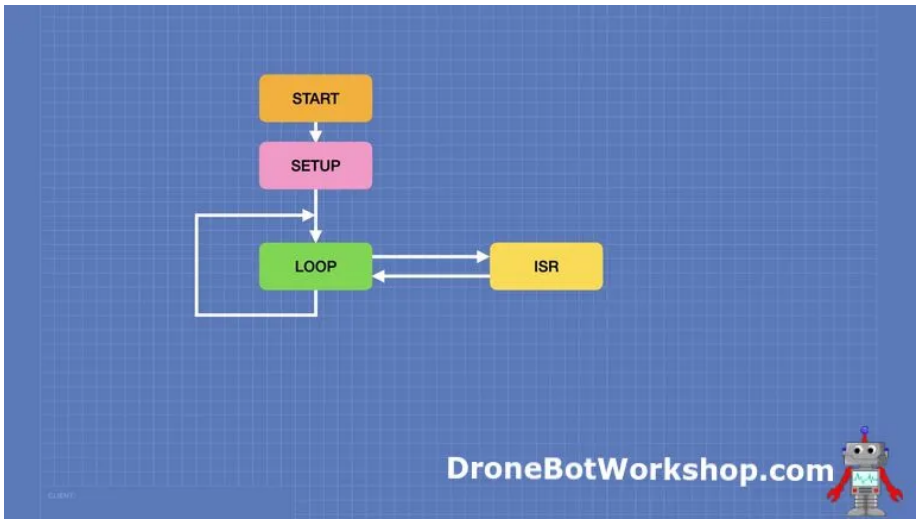


The sketch starts by including libraries (if required) and defining global variables and objects.

In the Setup function, we set PinModes, start objects and devices, and run any one-time code that we want to execute when the microcontroller is started.

Then we move on to the Loop. In the Loop, we run the body of our code, in sequence. Once we reach the bottom of the Loop, we start again from the top. We remain in the Loop until the microcontroller is reset, at which point we are back at the Start.

Now look at the same flowchart, only this time using an interrupt:



The execution of the sketch is identical to the original example, with the program remaining inside the Loop after running the Start and Setup procedures. But in addition, we have another box labeled “ISR”. This is the Interrupt Service Routine, and it will run whenever an interrupt event occurs.

So in our flowchart, we can see that the code execution branches out of the Loop and to the ISR. Once the ISR code is finished, the code execution is back at the Loop, in the same place that it branched off.

The ISR is only run when an interrupt occurs.

It is also possible to have a sketch that ONLY has an ISR, and does not use the Loop. In this case, nothing would run until an interrupt occur[illegible]'ll see several examples of this in a bit.

Interrupt Service Routines

An Interrupt Service Routine, or ISR, is essentially a function. However, unlike regular Arduino functions, you can't pass parameters to it, nor get any value returned from it.

Actually, ISR functions have a number of restrictions, most of them due to the same thing – they need to be fast. Very, very fast.

Think about it, you're literally interrupting a microcontroller that is doing its job, and a lot of that job involves timing things. You can't interrupt it for long, so there are many things that you can't do within an ISR:

- You can't use a delay() function.
- Not only that, but you also can't use the millis() function.
- No Serial library, so no printing to the serial monitor.
- Only use global variables, which should be declared volatile.

Despite these restrictions, an ISR can perform very useful work, like changing the value of one or more global variables that can then be read inside the Loop.

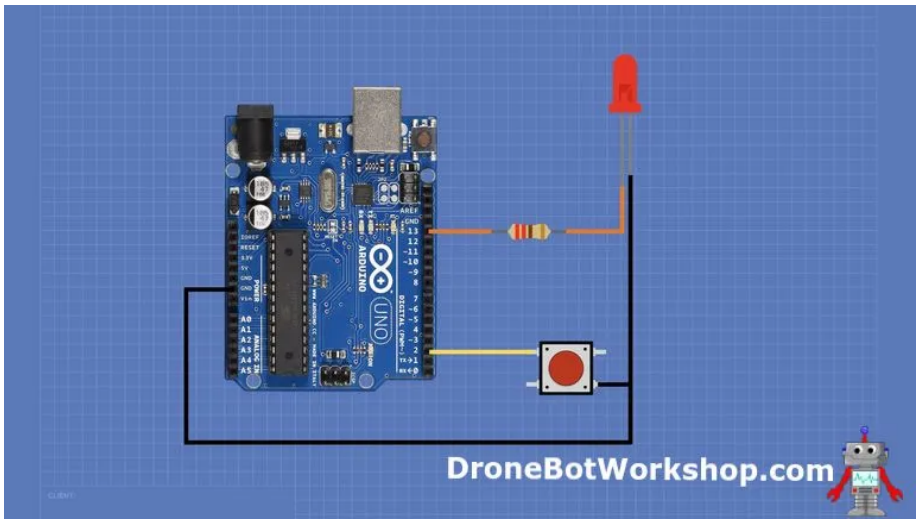
Just remember to do that useful work as quickly as possible!

Why use Interrupts?

To illustrate the value of using interrupts, we will run a very simple experiment that really only requires a pushbutton switch in addition to an Arduino. I'm also using an LED, but as it's hooked to pin 13 you could forgo it and just rely upon the built-in LED instead.

In our experiment, we will get the pushbutton to act as a toggle, turning on and off the LED alternately with each button press. It's a pretty common application, I'm sure you have seen it and probably have coded for it before.

Here is how we will hook it all up:



I used a 220-ohm dropping resistor for my red LED, but you could use any value from 150 to 470 ohms, and you can use any color of LED you like. Or just eliminate the LED and use the onboard one, as it is also connected to pin 13.

Now here is our equally simple sketch:

```
/*
  LED Toggle
  led-toggle.ino
  Use pushbutton switch to toggle LED

  DroneBot Workshop 2022
  https://dronebotworkshop.com
*/

// Define LED and switch connections
const byte ledPin = 13;
const byte buttonPin = 2;

// Boolean to represent toggle state

volatile bool toggleState = false;

void checkSwitch() {
  // Check status of switch
  // Toggle LED if button pressed

  if (digitalRead(buttonPin) == LOW) {
    // Switch was pressed
    // Slight delay to debounce
    delay(200);
    // Change state of toggle
    toggleState = !toggleState;
    // Indicate state on LED
    digitalWrite(ledPin,toggleState);
  }
}

void setup() {
      LED pin as output
  pinMode(ledPin, OUTPUT);
  // Set switch pin as INPUT with pullup
```

```
37    pinMode(buttonPin, INPUT_PULLUP);
38  }
39
40  void loop() {
41
42    // Check switch
43    checkSwitch();
44
45  }
```

We begin by declaring a couple of constant bytes for our switch and LED pins.

We also define a boolean called *togglestate*, this is what we will use to represent the current state of our toggle switch. This is initialized as false.

Next, we define a function called *checkSwitch*. It's a pretty simple function that checks the status of the pushbutton, if it is pressed then it inverts the current toggle value and then uses it to change the status of the LED.

In the Setup, we simply set the LED pin to be an output and the switch pin to be an input. We use the internal pull-up resistor for the input.

All we do in the Loop is to call the *checkSwitch* function, so we are always querying the switch status.

Load up the sketch and test it out. The LED should toggle on or off every time you press the switch.

Modifying (i.e. Breaking) our Sketch

Our toggle switch seems to work pretty well, and if all you wanted to do was to just have a toggle switch, then there is no reason you need to go any further.

But what if the toggle switch is just one component of a bigger design? Say a thermometer that used it to switch from Celsius to Fahrenheit. How easy would that be to build?

On the surface it seems pretty simple, just add the code for a DHT22 or similar sensor to the toggle code and use the toggle value to determine the temperature unit.

But, i ce, it can be a different story! To illustrate, let's make one slight modification to our code:

```
/*
  LED Toggle with Delay
  led-toggle-delay.ino
  Use pushbutton switch to toggle LED

  DroneBot Workshop 2022
  https://dronebotworkshop.com
*/

// Define LED and switch connections
const byte ledPin = 13;
const byte buttonPin = 2;

// Boolean to represent toggle state

volatile bool toggleState = false;

void checkSwitch() {
  // Check status of switch
  // Toggle LED if button pressed

  if (digitalRead(buttonPin) == LOW) {
    // Switch was pressed
    // Slight delay to debounce
    delay(200);
    // Change state of toggle
    toggleState = !toggleState;
    // Indicate state on LED
    digitalWrite(ledPin, toggleState);
  }
}

void setup() {
  // Set LED pin as output
  pinMode(ledPin, OUTPUT);
  // Set switch pin as INPUT with pullup
  pinMode(buttonPin, INPUT_PULLUP);
  // Setup Serial Monitor
  Serial.begin(9600);
}

void loop() {
  // Check switch
  checkSwitch();

  // Add a 5-second time delay
  Serial.println("Delay Started");
  delay(5000);
  Serial.println("Delay Finished");
  Serial.println(".............");
}
```

You'll notice two differences in the modified sketch:

- We set up the serial monitor and are printing to it in the Loop.
- We have added a 5-second delay to the bottom of the Loop.

Now, a 5-second delay is admittedly a silly thing to add to a sketc ut it in there to illustrate a point. It could just as easily be a reading of a DHT22, followed by a 2-second delay

while the sensor stabilizes. You get the idea, I'm adding a process that takes some time to complete to our Loop.

Load the sketch to the Arduino and try the switch again. I think you'll notice a difference.

You might think that you broke it, but really it's just degraded. Look at the serial monitor and observe, try pressing the pushbutton in that ever so brief period between delay events. You may get lucky and actually trigger the toggle.

The point I'm trying to illustrate, of course, is that if you have anything else in the Loop that might take up more than a few milliseconds of time, then polling the switch inside the Loop is not the best way to get a reading from it.

A better solution is to use Hardware Interrupts.

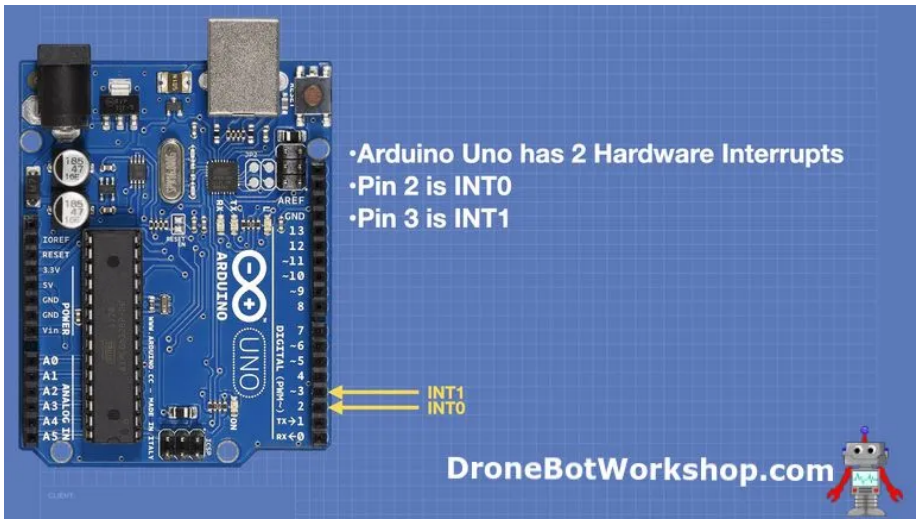
Hardware Interrupts

Hardware interrupts are external interrupts, and on most Arduino models are limited to specific pins. These pins are configured as inputs and can trigger hardware interrupts by manipulating their logic state.

These are probably the most common types of interrupts used by microcontroller experimenters, and we have used them here in the DroneBot Workshop on a number of occasions as well.

Arduino Hardware Interrupt Pins

On the Arduino Uno there are only two pins that are hardware interrupt capable:



- **Pin 2** – Interrupt vector 0
- **Pin 3** – Interrupt vector 1.

Not all Arduino boards are limited to 2 Hardware Interrupt pins, some Arduino boards have many more. The following chart shows the number of Hardware Interrupts on several common Arduino boards:

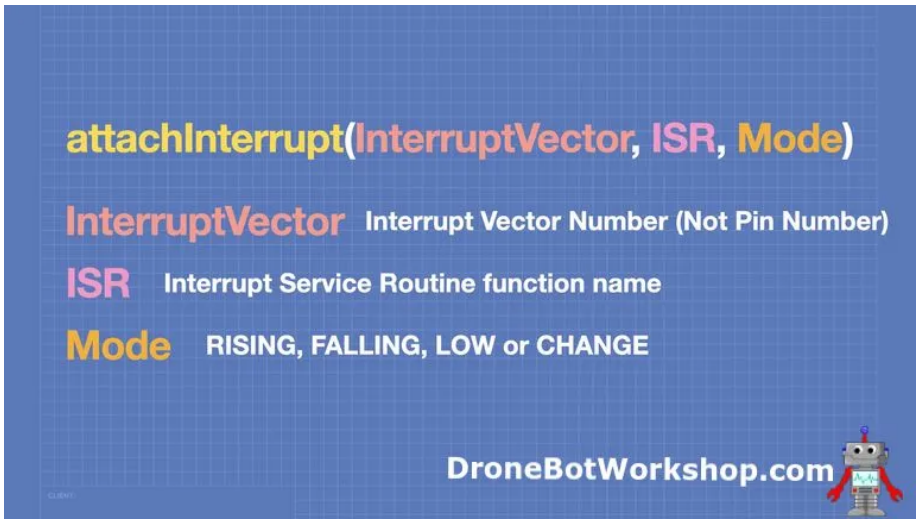
Working with Hardware Interrupts

Working with Hardware Interrupts is actually pretty simple, as you really only need to do two things:

- Write a function to use as your Interrupt Service Routine.
- Attach the function to the specific interrupt you want to use, and specify how to trigger it.

The ISR function should obey the rules about speed and using global variables, but otherwise, it's just a function. It can have any valid name that you want, but it can't have input parameters.

attachInterrupt Function



To "glue" your function to a specific interrupt, you'll be using the *attachInterrupt* function. This function has the following syntax and parameters:

- **Interrupt Vector** – the interrupt that you wish to use. Note that this is the internal Interrupt Vector number and NOT the pin number.
- **ISR** – The name of the Interrupt Service Routine function that you are gluing to the interrupt.
- **Mode** – How you want to trigger the interrupt.

For Mode, there are four selections:

- **RISING** – Triggers when an input goes from LOW to HIGH.
- **FALLING** – Triggers when an input goes from HIGH to LOW.
- **LOW** – Triggered when the input stays LOW.
- **CHANGE** – Triggers whenever the input changes state from HIGH to LOW or LOW to HIGH.

You would normally use *attachInterrupt* within the Setup function.

digitalPinToInterrupt Function

The Interrupt Vector parameter in the *attachInterrupt* function is not the same as the pin number, and it can vary between Arduino boards. A (better) way of obtaining this number is with the *digitalPinToInterrupt* function.

The function's name is also its description, it accepts a pin number and returns the Interrupt Vector number.

You can use *digitalPinToInterrupt* directly within the *attachInterrupt* function.

attachInterrupt(digitalPinToInterrupt(pin),ISR,Mode)

This is the preferred way of coding for Hardware Interrupts, as it makes the code portable between boards.

Rewriting our Sketch for Hardware Interrupts

Now that we know more about using Hardware Interrupts, let's modify our toggle and delay sketch to use them.

Here is an updated version of our sketch, using a Hardware Interrupt:

```
/*
  LED Toggle with Delay & Interrupt
  led-toggle-interrupt.ino
  Use pushbutton switch to toggle LED with interrupt

  DroneBot Workshop 2022
  https://dronebotworkshop.com
*/

// Define LED and switch connections
const byte ledPin = 13;
const byte buttonPin = 2;

// Boolean to represent toggle state

volatile bool toggleState = false;

void checkSwitch() {
  // Check status of switch
  // Toggle LED if button pressed

  if (digitalRead(buttonPin) == LOW) {
    // Switch was pressed
    // Change state of toggle
    toggleState = !toggleState;
    // Indicate state on LED
    digitalWrite(ledPin, toggleState);
  }
}

void setup() {
  // Set LED pin as output
  pinMode(ledPin, OUTPUT);
  // Set switch pin as INPUT with pullup
  pinMode(buttonPin, INPUT_PULLUP);

  // Attach Interrupt to Interrupt Service Routine
      Interrupt(digitalPinToInterrupt(buttonPin),checkSwi
}

```

```
41 void loop() {
42
43   // 5-second time delay
44   Serial.println("Delay Started");
45   delay(5000);
46   Serial.println("Delay Finished");
47   Serial.println(".............");
48 }
```

We begin similarly, by declaring the pin numbers for our LED and switch. We also use the same boolean for our toggle.

Note that the boolean is made volatile. This is important, as its value is being manipulated within the Interrupt Service Routine. Without the volatile statement, the Arduino IDE compiler may try and over-optimize the code and remove the variable.

Our *checkSwitch* function is almost identical to what it was before, the only difference is that we have removed the *delay* function. This is because we will be using *checkSwitch* as our Interrupt Service Routine, and we can't use a *delay* inside an ISR.

In Setup we do the usual *pinMode* commands, initialize the serial monitor, and then run *attachInterrupt* to attach the *checkSwitch* function to the Hardware Interrupt on pin 2. We use a FALLING mode, as we want to capture when we press (and ground) the switch.

All that we have in the Loop is the delay, which will now run continuously. Any switch activity will now be handled by the interrupt.

Load the sketch and play with the pushbutton while you observe both the LED and serial monitor. You should see that the toggle works despite the Loop always being inside a delay.

As you can see, Hardware Interrupts are a much more efficient method of capturing switch inputs.

Pin Change Interrupts

Pin Change Interrupts are another form of a hardware interrupt. Unlike the interrupts we have just used, they are not restricted to specific pins, all the pins can be used for Pin Change Interrupts.

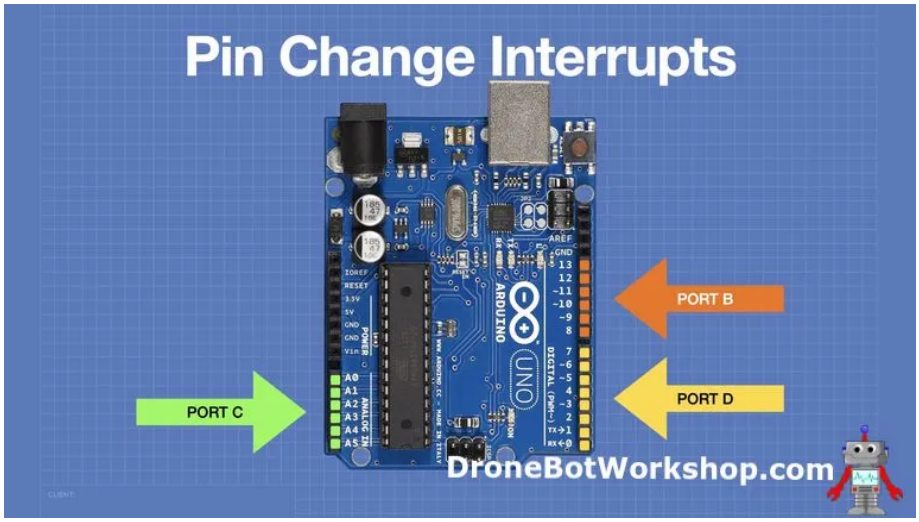
The c hat Pin Change Interrupts are grouped into Ports, and all the pins on the same port create the same Pin Change

Interrupt. This is fine if you are only using one, otherwise, you'll need to figure out which pin caused the interrupt.

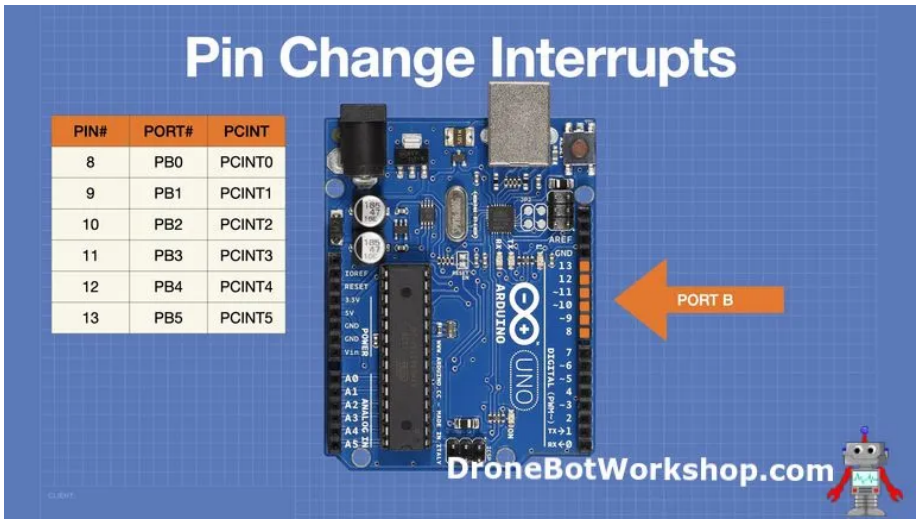
Pin Change Interrupts are restricted, as their name might imply, to only monitoring a CHANGE of logic state. So a switch press will generate two interrupts, one when the switch is pressed and a second one when it is released. If you need to know if the interrupt was caused by a HIGH or LOW input, then you'll have to figure that out yourself.

Pin Change Ports

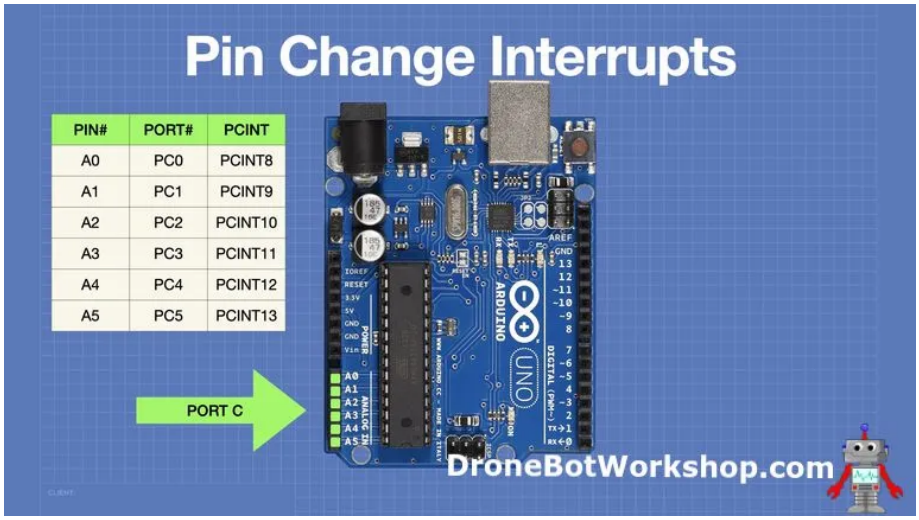
Almost every pin on the ATmega328 chip that the Arduino Uno (and other boards) are based upon can support 24 Pin Change Interrupts. That includes the two pins used for the 16MHz crystal oscillator.



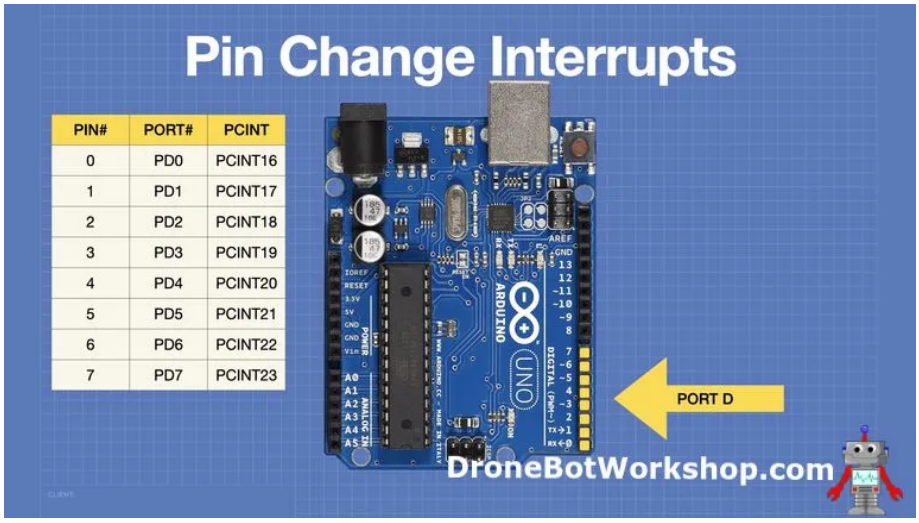
On the Arduino Uno, 20 pins are available for Pin Change Interrupts, and they are divided into three ports.



Pins 8 through 13 are on Port B.



Pins A0 to A5 are on Port C.



Pins 0 to 7 are on Port D.

Working with Pin Change Interrupts

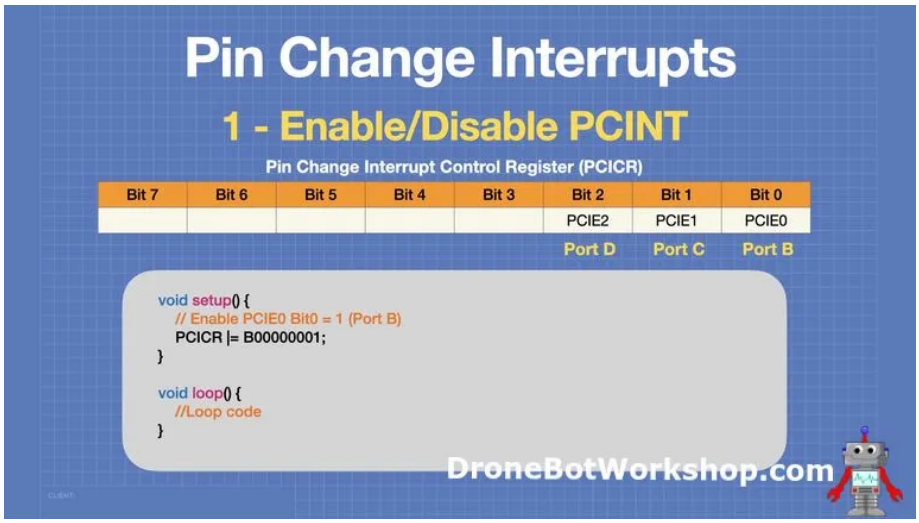
In order to use Pin Change Interrupts, you'll need to do the following:

- Determine which pin(s) that you want to use. This will also tell you which port(s) you'll need to use.
- Enable the port(s) that you need.
- Enable the pin(s) within that port that must be enabled for interrupts.
- Edit the appropriate Interrupt Service Routine(s). If you are using more than one pin on the same port, then the ISR will need to be able to determine which pin caused the interrupt.

Let's look at those steps in a bit more detail. For our example we will just use an individual pin, later we will run a sketch that uses two pins on the same port.

Select the Port

The first step is to enable the appropriate port, which you'll determine based on the pin number. To enable the port, you will use the *Pin Change Interrupt Control Register* or PCICR.

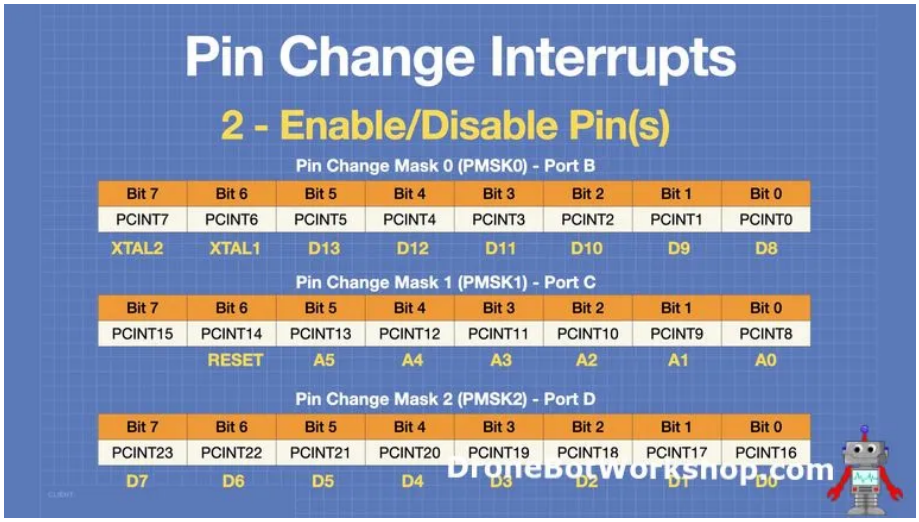


The PCICR has three bits of interest, Bit 0, Bit 1, and Bit 2. Each bit is associated with one of the ports, and setting it to 1 will enable the port.

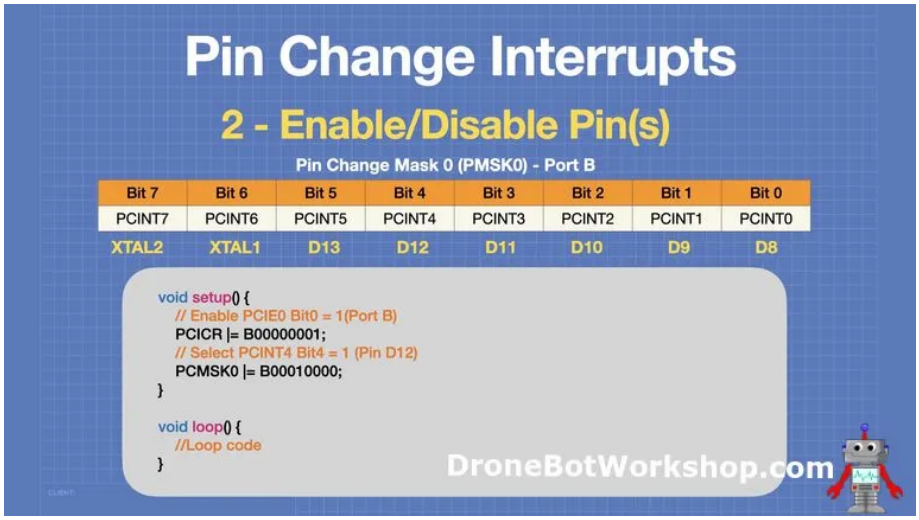
In Setup, you'll write a binary number to the PCICR to set the appr bit(s) to 1. You can, of course, enable more than one port.

Enable/Disable the Pins on the Port

After enabling the port, you'll need to enable the pin(s) that you want to use for Pin Change Interrupts. You do that by modifying the *Pin Change Mask* for your selected port.



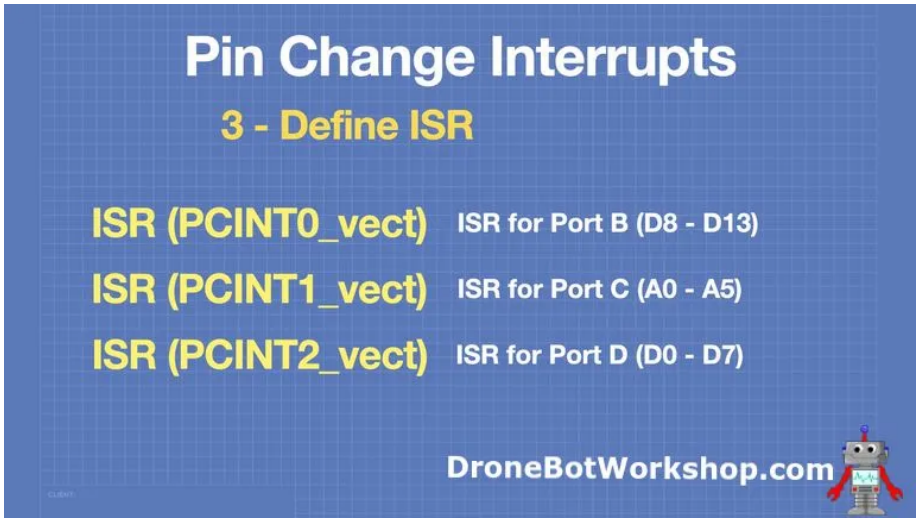
There are three Pin Change Masks, and each one can enable or disable 8 pins. To enable a pin, you write a "1" to it. You can enable as many pins as you require, remember you'll need to find a way to differentiate between them in your Interrupt Service Routine.



Interrupt Service Routine

Unlike Hardware Interrupts, you don't just create an Interrupt Service Routine and give it any random name. With Pin Change Interr[illegible]he ISR has already been defined for you, so you'll need to use the correct one for your port.

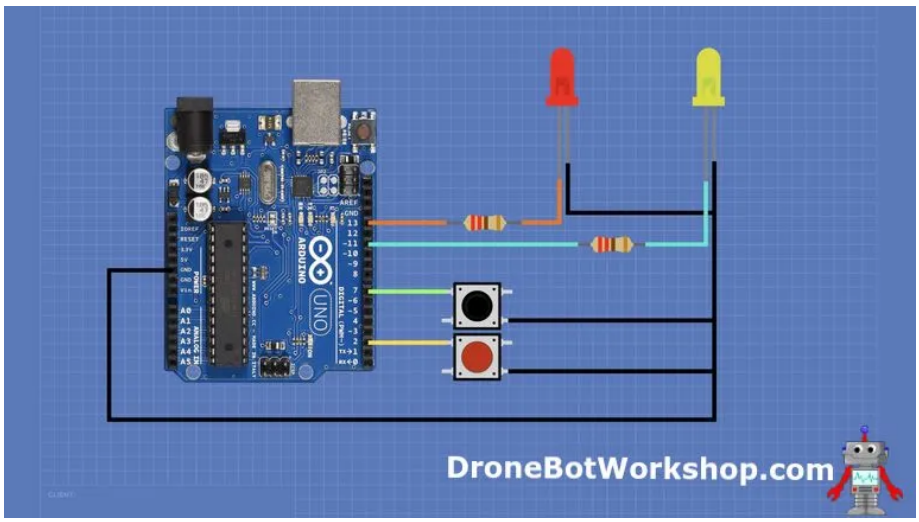
There are three ports and thus three ISRs, which are named as shown here:



All the same rules regarding Interrupt Service Routines apply to the ISR you use with Pin Change Interrupts. Keep it short and use volatile global variables.

Experimenting with Pin Change Interrupts

In order to run the next couple of examples, we will need to add another pushbutton and LED to our Arduino. The final hookup is shown here:



So we now have:

- utton on pin D2.
- A pushbutton on pin D7.

- An LED on pin D11.
- An LED on pin D13.

Note that the two pushbutton switches, on D2 and D7, are on the same port, port D.

Pin Change Interrupt Example 1 – Simple Interrupt

The first example we will run will illustrate how to code for Pin Change Interrupts. We will only be using one switch and one LED in this experiment. Note that our switch is on pin D7, which is not a Hardware Interrupt pin. That doesn't matter, as we'll be sensing it with a Pin Change Interrupt!

We will be toggling the state of an LED every time we get an interrupt on pin D7. One important thing to keep in mind is that we will be sensing an input CHANGE of state, so a pushbutton will produce two interrupts – one when pressing it down and another one when releasing it.

This will make our LED operate a lot like it was just wired in series with the switch!

Here is our sketch:

```
/*
  Pin Change Interrupt Test
  pin-change-test.ino
  Demonstrates use of Pin Change Interrupt
  Input on D7, LED on D13

  DroneBot Workshop 2022
  https://dronebotworkshop.com
*/

// LED and switch
const byte ledPin = 13;
const byte buttonPin = 7;

// Boolean to represent toggle state
volatile bool togglestate = false;

void setup() {

  // Set LED as output
  pinMode(ledPin, OUTPUT);

  // Set switch as input with pullup
      e(buttonPin, INPUT_PULLUP);


```

```
  // Enable PCIE2 Bit3 = 1 (Port D)
  PCICR |= B00000100;
  // Select PCINT23 Bit7 = 1 (Pin D7)
  PCMSK2 |= B10000000;

}

void loop() {
  // No code in Loop
}

ISR (PCINT2_vect)
{
  // Interrupt for Port D
  // Invert toggle state
  togglestate = !togglestate;
  // Indicate state on LED
  digitalWrite(ledPin, togglestate);


}
```

We start by defining pins for the switch and LED.

Next we create a volatile boolean to represent the toggle state.

In Setup, we use the *pinMode* function to define our inputs and outputs.

And then we set up our Pin Change Interrupt. We modify the PCICR register to let it know we want to use port D, and we modify port D's mask PCMSK2 to set pin D7 as an interrupt input.

We don't have any code in the Loop, as we are doing everything with an interrupt service routine.

The Interrupt Service Routine is next, and as we are using port D we use *ISR (PCINT2_vect)*. In the ISR we flip the state of the *toggleState* variable and use it to drive the LED.

Load the code and try it out. The LED should come on when you hold the switch down and go off when you release it. Which, of course, is a real overkill using a microcontroller, but it does demonstrate Pin Change Interrupts!

Pin Change Interrupt Example 2 – Multiple Interrupts on the Same Port

In the previous example, we were just able to act upon the Pin Chan[illegible]rupt, as only one pin on our port was enabled for

interrupts. There was no question about what had caused the interrupt.

But how do you deal with it when you enable two or more pins on the same port for Pin Change Interrupts? You will need to determine who caused the interrupt so that you can act accordingly.

In this example, we will have two pushbuttons and two LEDs. Each pushbutton will act as a toggle for its respective LED. That means that we need to know two things when we get an interrupt:

- What pin caused the interrupt?
- Was it LOW or HIGH?

We will answer both of those questions within our sketch:

```
/*
  Multiple Pin Change Interrupt Demo
  pin-change-multiple-test.ino
  Demonstrates Pin Change Interrupts with two on same port
  Inputs on D2 & D7, LEDs on D11 and D13

  DroneBot Workshop 2022
  https://dronebotworkshop.com
*/

// LEDs and switchs
const byte ledPin1 = 11;
const byte ledPin2 = 13;
const byte buttonPin1 = 2;
const byte buttonPin2 = 7;

// Booleans for input states
volatile bool D2_state = LOW;
volatile bool D7_state = LOW;

void setup() {

  // Set LEDs as outputs
  pinMode(ledPin1, OUTPUT);
  pinMode(ledPin2, OUTPUT);

  // Set switches as inputs with pullups
  pinMode(buttonPin1, INPUT_PULLUP);
  pinMode(buttonPin2, INPUT_PULLUP);

  // Enable PCIE2 Bit3 = 1 (Port D)
  PCICR |= B00000100;
  // Enable PCINT18  & PCINT23 (Pins D2 & D7)
  PCMSK2 |= B10000100;

}

void loop() {

  // Loop code
```

```
41
42 }
43
44 ISR (PCINT2_vect)
45 {
46   // Port D Interrupt occured
47
48   // Check if this was D2
49   if (digitalRead(buttonPin1) == LOW) {
50     //Pin D2 triggered the ISR on a Falling pulse
51     D2_state = !D2_state;
52     //Set LED 1 to state of D2_state boolean
53     digitalWrite(ledPin1, D2_state);
54   }
55
56   // Check if this was D7
57   if (digitalRead(buttonPin2) == LOW) {
58     //Pin D7 triggered the ISR on a Falling pulse
59     D7_state = !D7_state;
60     //Set LED 2 to state of D7_state boolean
61     digitalWrite(ledPin2, D7_state);
62   }
63
64 }
```

The sketch begins as you would expect, we define a bunch of variables for the LEDs and switches and two booleans, one for each toggle state.

In the Setup, we set the *pinModes* for the switches and LEDs, again using the internal pull-up resistors for the inputs.

We then enable Port D, as we did in the previous sketch, by writing a "1" to bit 2 of the PCICR register.

Next, we write a "1" to both bits 7 and 2 of the PCMSK2 mask, to let it know that pins D7 and D2 are to be treated as Pin Change Interrupts.

Again there is no code in the Loop, everything is being done in the Interrupt Service Routine.

That routine is ISR (PCINT2_vect), the ISR for Port D., which is the same ISR we used in the previous sketch. Only this time, we will need to figure out if it was pin D2 or D7 that caused the interrupt.

We are looking for a LOW condition in this sketch, as we want to toggle the button when the switch is pressed, but not when it is released. Also, we use a couple of if statements along with a *digitalRead* to check each input and determine if it is currently LOW. If it is, then we toggle the respective boolean and use it to control the LED.

Load the sketch and give it a test. The pushbutton tied to pin D2 should control the LED on pin D11, and the button on D7 should work with the LED on pin D13. You should be able to toggle each of them independently.

Pin Change Interrupts can be very useful when scanning a group or matrix of switches, or when you need an external interrupt but haven't got a Hardware Interrupt pin to spare.

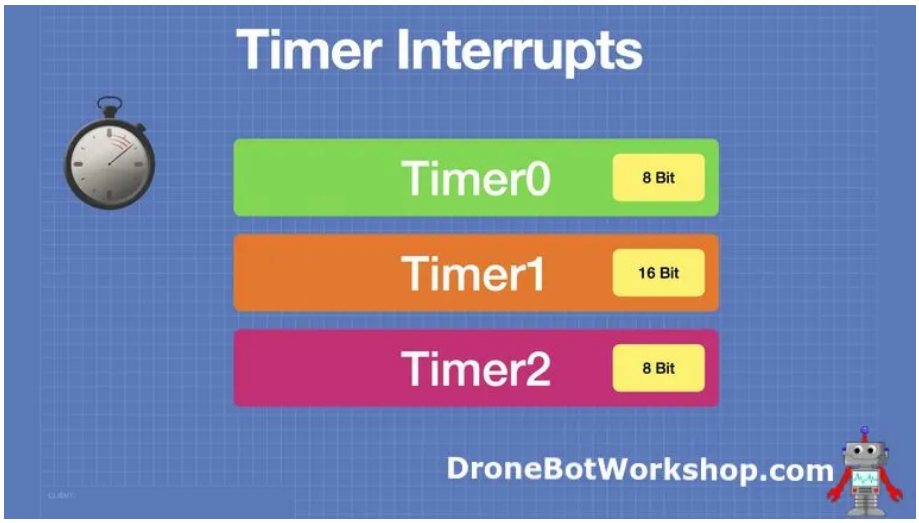
Timer Interrupts

Timer interrupts don't use external signals. Instead, these interrupts are generated in software, and their timing is based upon the Arduino Uno's 16 MHz clock oscillator.

You've likely been using Timer Interrupts without realizing it, as several popular libraries, such as the Servo and Tone libraries, use timer interrupts internally. Bear in mind that if you are using a library that makes use of the timers, you'll need to know that so that you don't write conflicting code.

Arduino Uno Timers

The Arduino Uno has three internal timers, Timer0, Timer1, and Timer2.

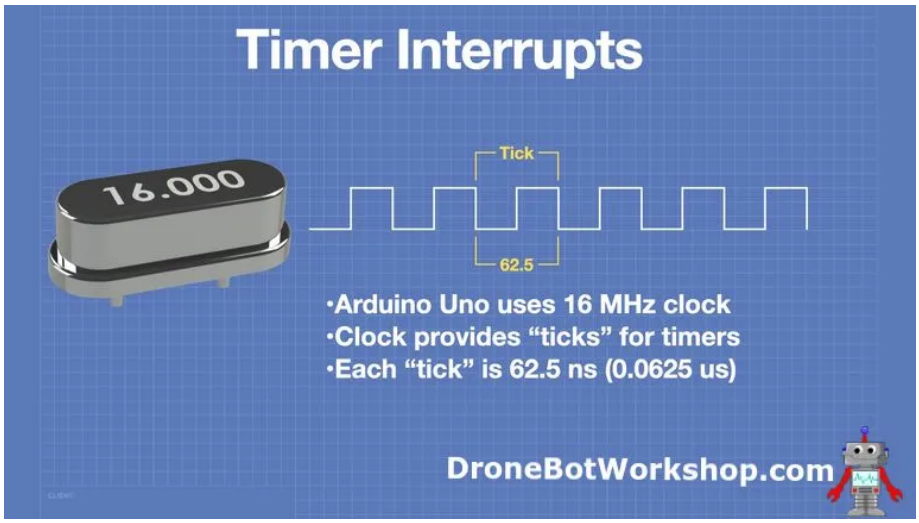


These timers are not the same, as Timer1 is a 16-bit timer, whereas the other two timers are just 8-bit timers. The number of bits determines the maximum number that the timer can count to, 256 for the 8-bit timers and 65,536 for the 16-bit one.

The value in these timers is incremented by either the clock frequency or a fraction of it. You can use software to determine the count you wish to set the interrupt to trigger at, you can also trigger an interrupt when the timer overflows.

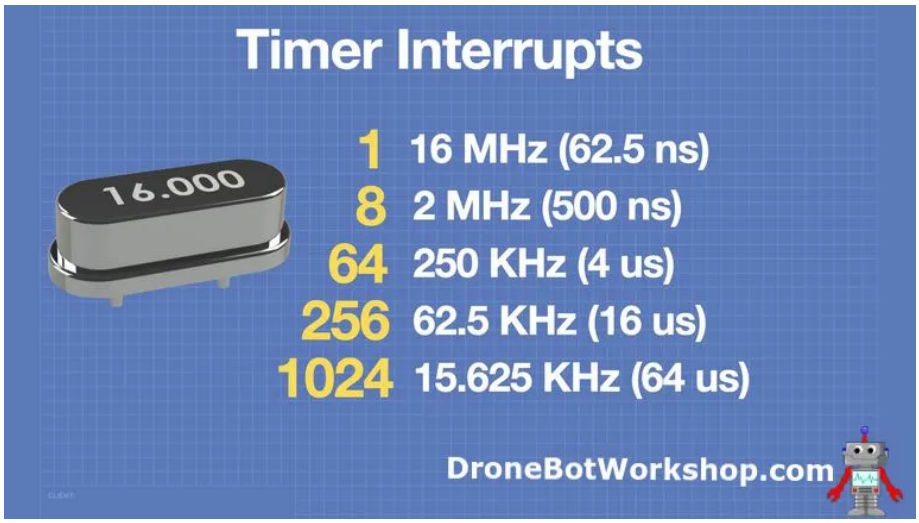
Dividing the Clock Frequency

The Timers are clocked by the 16 MHz oscillator which is internal to the ATmega328.

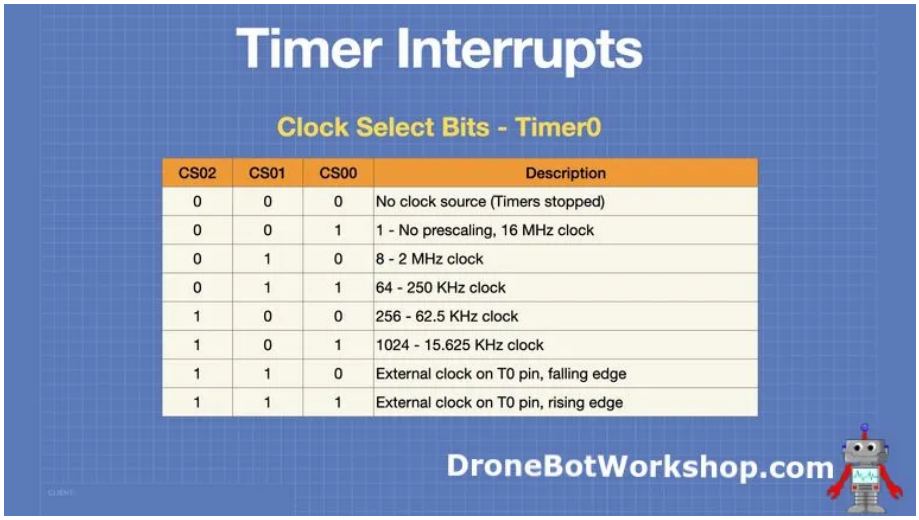


Every cycle of the clock is a timer "tick", so at 16 MHz a "tick" would be a period of 62.5 nanoseconds. This is a pretty short period, and for many timing applications, it would be too short to be of much practical use.

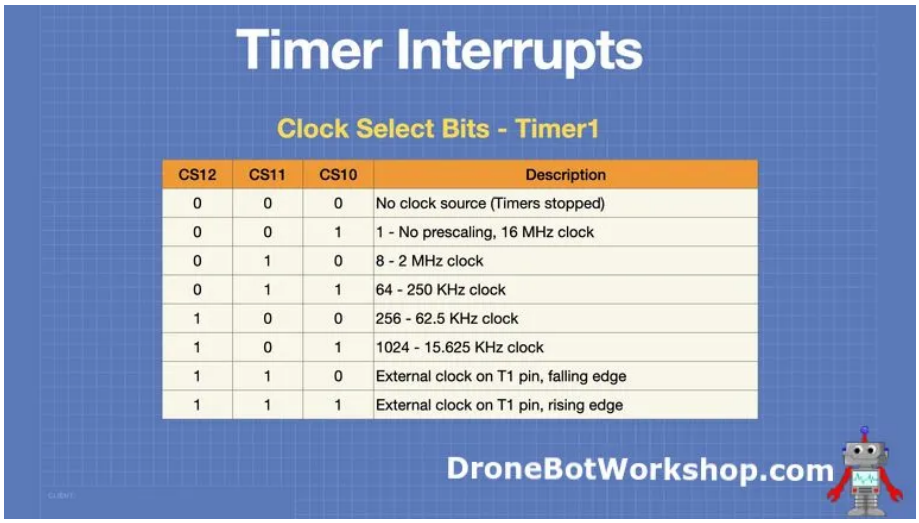
To slow down the clock signal, the ATmega328 has a "Prescaler", essentially a divider for the clock frequency. The Prescaler can divide the clock down to a more manageable lower frequency, you can select from a number of common divisions to create pulses as long as 64us.



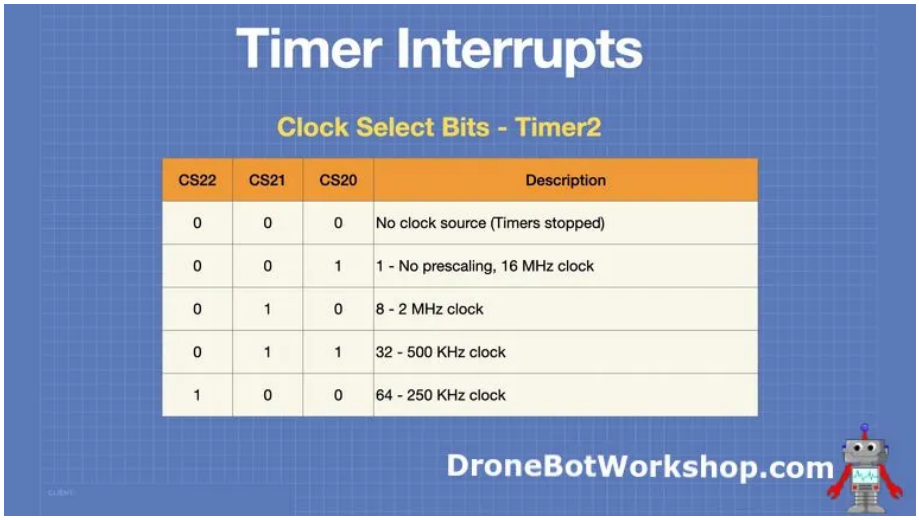
Each timer has three Clock Select Bits, and the value of these bits can determine the Prescaler settings, as well as the timing source. You can also stop the clock entirely by setting all the Clock Select Bits to zero.



Timer0, an 8-bit Timer, uses bits CS01, CS02, and CS03.



Timer1, the 16-bit timer, uses bits CS10, CS11, and CS12.



Timer2, another 8-bit timer, uses bits CS20, CS21, and CS22.

Using Timer Interrupts

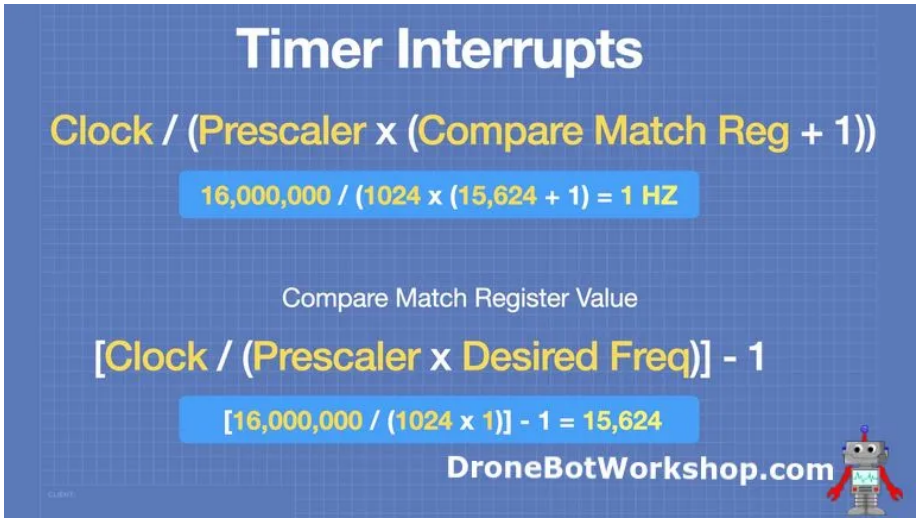
Timer Interrupts can be operated in a couple of different modes, including *Compare Match Mode* and *Overflow Mode*.

In Compare Match Mode, you place a counter value into a Compare Match Register. The timer interrupt is generated when the timer counter matches the value in that register.

In Overflow Mode the timer interrupt is generated when the timer reaches the end of its count, an interrupt is generated and the timer resets to zero and begins counting again.

By combining the Compare Match Register with a Prescaler you can get pretty well any timing period you want, assuming it is in the range of your timer (the 8-bit timers will only divide by a maximum of 255).

The formula for determining the output frequency, and thus the period, of a timer, is as follows:



Clock/(Prescaler x (Compare Match Register + 1)) = Frequency

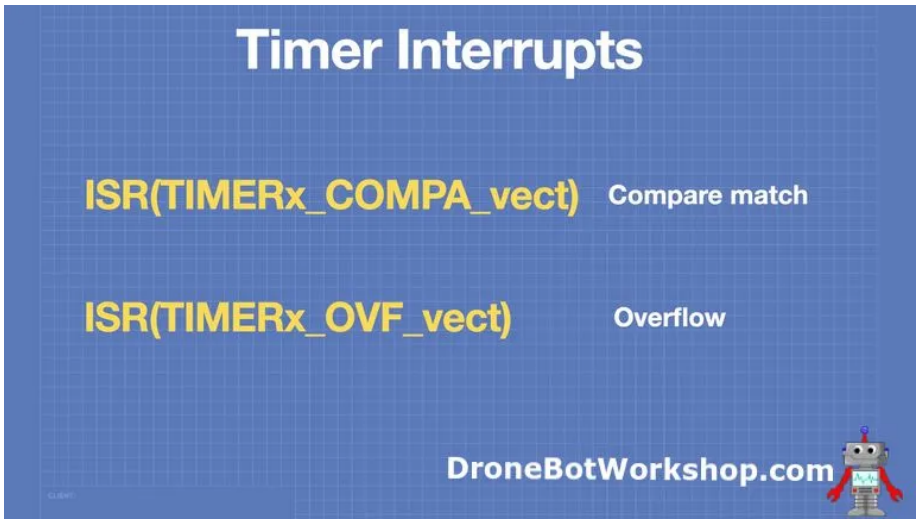
If you know the frequency that you want and want to determine the Compare Match Register value, you can rewrite the formula as follows:

[Clock/(Prescaler x Frequency)] -1 = Compare Match Register

Using the above formulas we can calculate that to get a 1 Hz output we can use a Prescaler of 1024 and a Compare Match Register of 15,624. As this value exceeds 255 we would be limited to using Timer1, the 16-bit timer.

Timer Interrupt Service Routines

As with the Pin Change Interrupts the names of the interrupt service routines for Timer Interrupts have already been determined for you.



Each timer has two ISRs associated with it, one for Compare Match Mode and the other for Overflow Mode.

In the above diagram, the"x" after the word "TIMER" should be replaced by the timer number, such as 0, 1, or 2.

Simple Timer Example

This very simple timer example sets up a timer for a 2 HZ output, or twice a second. We then use the timer to control an LED, so essentially we are building the Blink sketch!

We will be using an Arduino Uno and an LED (with dropping resistor) on pin 13. If you wish, you can just grab one of the previous experiments, or just use the built-in LED on the Arduino board.

Here is the sketch that we will be using:

```
/*
  Arduino Timer Interrupt Flash Demo
  timer-int-flash.ino
  Flash LED using Timer1

  DroneBot Workshop 2022
  https://dronebotworkshop.com
*/

// Define the LED pin
#define ledPin 13

// Define timer compare match register value
int timer1_compare_match;


i      (1_COMPA_vect)
// Interrupt Service Routine for compare mode
```

```
{
  // Preload timer with compare match value
  TCNT1 = timer1_compare_match;

  // Write opposite value to LED
  digitalWrite(ledPin, digitalRead(ledPin) ^ 1);
}

void setup()
{
  // Set LED as output
  pinMode(ledPin, OUTPUT);

  // Disable all interrupts
  noInterrupts();

  // Initialize Timer1
  TCCR1A = 0;
  TCCR1B = 0;

  // Set timer1_compare_match to the correct compare match
  // 256 prescaler & 31246 compare match = 2Hz
  timer1_compare_match = 31249;

  // Preload timer with compare match value
  TCNT1 = timer1_compare_match;

  // Set prescaler to 256
  TCCR1B |= (1 << CS12);

  // Enable timer interrupt for compare mode
  TIMSK1 |= (1 << OCIE1A);

  // Enable all interrupts
  interrupts();
}



void loop()
{

}
```

In this sketch, we are using the Compare Match Mode, so after we define an LED pin, we also create an integer to hold the compare match register value.

Next, our Interrupt Service Routine. As we are using Compare Match Mode on Timer1, we will use *ISR(TIMER1_COMPA_vect)*.

In the ISR we do two things:

- Preload the timer with the compare match value, to start the cycle again
- Flip the value of the LED.

In Setup, we set our LED pin as an output. We then temporarily disable all interrupts, to prevent one from coming along while we are still setting things up.

We initialize Timer1 with two commands and then set up our compare match value. Since we are looking to achieve 2 Hz, we calculate that 31246 is a good value if we use a Prescaler of 256.

We then preload our timer with the compare match value, set our Prescaler for 256, and enable interrupts in compare match mode.

Finally, we re-enable all the interrupts.

Load the sketch and observe the LED flashing. After the thrill of that is gone, experiment with different Prescaler and Compare Match values.

Conclusion

Interrupts are a great way to build projects that require precise timing or a responsive user interface. While most of us have dabbled with Hardware Interrupts, many people shy away from Pin Change or Timer interrupts, which is a shame as they are so useful and the binary coding isn't really that difficult once you understand the use of the registers.

I'd encourage you to incorporate interrupts into your next design. Remember, in some cases, it isn't rude to be interrupted!

Parts List

Here are some components that you might need to complete the experiments in this article. Please note that some of these links may be affiliate links, and the DroneBot Workshop may receive a commission on your purchases. This does not increase the cost to you and is a method of supporting this ad-free website.

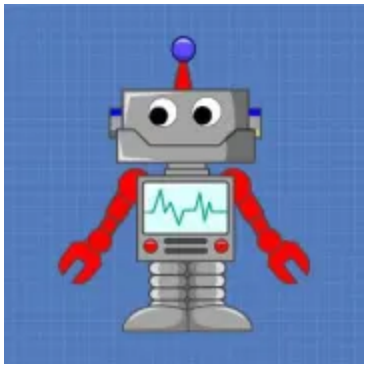
COMING SOON!

Resources

Code Samples – All the code used in this article, packed up nicely in a ZIP file.

attachInterrupt – Reference for the attachInterrupt function used with Hardware Interrupts.

Summary

Article Name	Using Arduino Interrupts - Hardware, Pin Change and Timer
Description	Interrupts are the key to construction projects with responsive user interfaces or that have precision timing requirements. Today we will learn how to use Hardware, Pin Change, and Timer Interrupts with the Arduino Uno.
Author	DroneBot Workshop
Publisher Name	DroneBot Workshop
Publisher Logo	

Tagge Arduino Tutorial