# Using Interrupts on Arduino

August 12, 2015 by Nash Reilly (/author/nash-reilly)

Streamline your Arduino code with Interrupts - the simple way of reacting to real-time events!

## Recommended Level

Intermediate

## We Interrupt This Broadcast...

As it turns out, there's a great (and underutilized) mechanism built into all Arduinos that is ideal for monitoring these kinds of real-time events. This mechanism is called an Interrupt. An Interrupt's job is to make sure that the processor responds quickly to important events. When a certain signal is detected, an Interrupt (as the name suggests) interrupts whatever the processor is doing, and executes some code designed to react to whatever external stimulus is being fed to the Arduino. Once that code has wrapped up, the processor goes back to whatever it was originally doing as if nothing happened!

What's awesome about this is that it structures your system to react quickly and efficiently to important events that aren't easy to anticipate in software. Best of all, it frees up your processor for doing other stuff while it's waiting on an event to show up.

## Button Interrupts

Let's start with a simple example - using an Interrupt to monitor a button press. To start, we'll take a sketch you've likely seen before - the "Button" example sketch included with all Arduinos. (You can find this in the "Examples" sketchbook. Check out "File > Examples > Digital > Button".)

```
const int buttonPin = 2;     // the number of the pushbutton pin
const int ledPin =  13;      // the number of the LED pin

// variables will change:
int buttonState = 0;         // variable for reading the pushbutton status

void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
}

void loop() {
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

**Download Code (/login/?redirect=https://www.allaboutcircuits.com/technical-arti**

What you're seeing here is nothing shocking or amazing - all the program is doing, over and over again, is running through `loop()`, and reading the value of `buttonPin`. Suppose for a moment that you wanted to do something else in `loop()` - something besides just reading a pin. That's where Interrupts come in. Instead of just watching that pin all the time, we can farm the work of monitoring that pin to an Interrupt, and free up `loop()` to do whatever we need it to do in the meantime! The new code would look something like this:

```
const int buttonPin = 2;     // the number of the pushbutton pin
const int ledPin =  13;      // the number of the LED pin

// variables will change:
volatile int buttonState = 0;         // variable for reading the pushbutton status

void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
  // Attach an interrupt to the ISR vector
  attachInterrupt(0, pin_ISR, CHANGE);
}

void loop() {
  // Nothing here!
}

void pin_ISR() {
  buttonState = digitalRead(buttonPin);
  digitalWrite(ledPin, buttonState);
}
```

**Download Code (/login/?redirect=https://www.allaboutcircuits.com/technical-arti**

## Loops and Interrupt Modes

You'll notice a few changes here. The first, and most obvious of these, is that `loop()` doesn't contain any instructions! We can get away with this because all of the work that was previously done by an `if/else` statement is now handled by a new function, `pin_ISR()`. This type of function is called an _interrupt service routine_ - its job is to run quickly and handle the interrupt and let the processor get back to the main program (i.e. the contents of `loop()`). There are a few important things to consider when writing an interrupt service routine, which you can see reflected in the code above:

- ISRs should be short and sweet. You don't want to derail the main loop for too long!
- There are no input variables or returned values. All changes have to be made on global variables.

You're probably wondering - how do we know when an interrupt gets run? What triggers it? The third function in the `setup()` routine is what sets up the interrupt for the whole system. This function, `attachInterrupt()`, takes three arguments:

1. The *interrupt vector*, which determines what pin can generate an interrupt. This isn't the number of the pin itself - it's actually a reference to where in memory the Arduino processor has to look to see if an interrupt occurs. A given space in that vector corresponds to a specific external pin, and not all pins can generate an interrupt! On the Arduino Uno, pins 2 and 3 are capable of generating interrupts, and they correspond to interrupt vectors 0 and 1, respectively. For a list of what pins are available as interrupt pins, check out the Arduino documentation (https://www.arduino.cc/en/Reference/attachInterrupt) on `attachInterrupt()`.

2. The *function name of the interrupt service routine* - this determines the code that gets run when the interrupt condition is met.

3. The *interrupt mode*, which determines what pin action triggers an interrupt. The Arduino Uno supports four interrupt modes:

* `RISING`, which activates an interrupt on a rising edge of the interrupt pin,

* `FALLING`, which activates on a falling edge,

* `CHANGE`, which responds to any change in the interrupt pin's value,

* `LOW`, which triggers any time the pin is a digital low.

Just to recap - our setting of `attachInterrupt()` is setting us up to monitor interrupt vector 0/pin 2, to respond to interrupts using `pin_ISR()`, and to call `pin_ISR()` whenever we see any change of state on pin 2.

## Volatile - Do Not Shake!

One more quick thing to point out - our ISR uses the variable `buttonState` to store pin state. Check out the definition of `buttonState` - instead of type `int`, we've defined it as type `volatile int`. What's the deal here? `volatile` is a C keyword applied to variables. It means that the value of that variable is not entirely within a program's control. It reflects that the value of `buttonState` could change, and change on something that the program itself can't predict - in this case, user input.

One more useful thing that the `volatile` keyword does is protect us from any accidental compiler optimization. Compilers, as it turns out, have a few purposes in addition to turning your source code into a machine executable. One of their tasks is to trim unused source code variables out of machine code. Since the variable `buttonState` is not used or called directly in the `loop()` or `setup()` functions, there is a risk that the compiler might remove it as an unused variable. Obviously, this is wrong - we need that variable! The `volatile` keyword has the side effect of telling the compiler to cool its jets and hang on to that variable - it's not a fat finger error!

(Aside: pruning unused variables from code is a feature, not a bug, of compilers. People will occasionally leave unused variables in source code, which takes up memory. This isn't such a big deal if you were writing a C program on a computer, with gigabytes of RAM. On an Arduino, however, RAM is limited, and you don't want to waste any! Even C programs on computers will do this as well, with tons of system memory available. Why? The same reason people clean up campgrounds - it's good practice to not leave garbage behind!)

## Wrapping Up

Interrupts are a simple way to make your system more responsive to time sensitive tasks. They also have the added benefit of freeing up your main `loop()` to focus on some primary task in the system. (I find that this tends to make my code a little more organized when I use them - it's easier to see what the main chunk of code was designed for, while the interrupts handle periodic events.) The example shown here is just about the most basic case for using an interrupt - you can use them for reading an I2C device, sending or receiving wireless data, or even starting or stopping a motor.

Got any cool projects with interrupts? Leave us a comment below to let us know!