Arduino Interrupts Tutorial

What are Arduino Interrupts? How to use them? What should you know about them?

In this Arduino Interrupts tutorial I'll show you an example of when you can use interrupts and how to handle them. I'll also give you a list of important points you should pay attention to, because, as you'll see, interrupts are something you should handle with care.

Table of Contents

What is an Interrupt pin?

A real life analogy

Example 1

Let's use a real life analogy. Imagine you're waiting for an important email. You don't know when it will arrive, but you want to make sure you read it as soon as it arrives in your mailbox.

The most basic solution is to frequently check your mailbox – let's say, every 5 minutes – so you're sure the maximum delay between the reception of the email, and you reading it, is 5 minutes. But this is really not an ideal solution. First, you'll spend all your time refreshing your mailbox and won't do any productive thing in the meantime. And second, this is relatively inefficient. When the email arrives, you'll have up to 5 minutes delay before you read it.

This technique is called "**polling**". **At a given frequency, you're polling the state of something to see if a new information arrived.** At a human scale you see that it's completely not worth it.

The other possible way to do that is to use interrupts. For us humans, this means turning on notifications. As soon as the email has arrived, you will get a popup on your phone/computer saying that the email is here. You can now check your email, and the delay between the reception and you reading the email is basically zero.

Let's add more details to this analogy: the email you're about to receive contains a special offer to get a discount on a given website – and this offer is available only for 2 minutes. **If you use the "polling" technique, there is a chance that you miss some data** (in this example, you'll miss the discount). **With interrupts, you can be sure you won't miss it.**

Example 2

Another example: you're waiting to talk to the postman about something. You now he will arrive between 9am and 11am. First option – polling – you can keep going to your door to check if he has arrived. But maybe you'll miss him, because you can't always be at your window looking at the street.

Second option – interrupts – you put a note on your door saying "Dear Mr. Postman, please ring the bell when you see this". As soon as the postman arrives, he will ring the bell and you won't miss him.

In both scenarios, you stop your current action. That's why it's called an interruption. You have to stop what you're doing to handle the interruption, and only after you're done with it, you can resume your action.

Interrupts on Arduino

Arduino Interrupts work in a similar way.

For example, if you are waiting for a user to press on a push button, you can either monitor the button at a high frequency, or use interrupts. With interrupts, you're sure that you won't miss the trigger.

The monitoring for Arduino Interrupts is done by hardware, not software. As soon as the push button is pressed, the hardware signal on the pin triggers a function inside the Arduino code. This stops the mains execution of your program. After the triggered function is done, the main execution resumes.

Note that for the real life analogies above, interrupts make much more sense than the polling technique. However I want to point that sometimes, polling can be a better choice. At human scale, interrupts make much more sense. At a micro-controller scale, where the frequency of execution is much higher, sometimes it becomes complicated than that and the choice is not always obvious. We'll discuss more about it later in this post.

Arduino Interrupts Pins

Arduino Interrupts Pins are using digital pins. However, usually you can't use all available digital pins. Only some of them have the functionality enabled.

Here are the pins you can use for interrupts on the main Arduino boards:

Arduino Board	**Digital Pins for Interrupts**
Arduino Uno, Nano, Mini	2, 3
Arduino Mega	2, 3, 18, 19, 20, 21
Arduino Micro, Leonardo	0, 1, 2, 3, 7
Arduino Zero	All digital pins except 4

Arduino Board	Digital Pins for Interrupts
Arduino Due	All digital pins

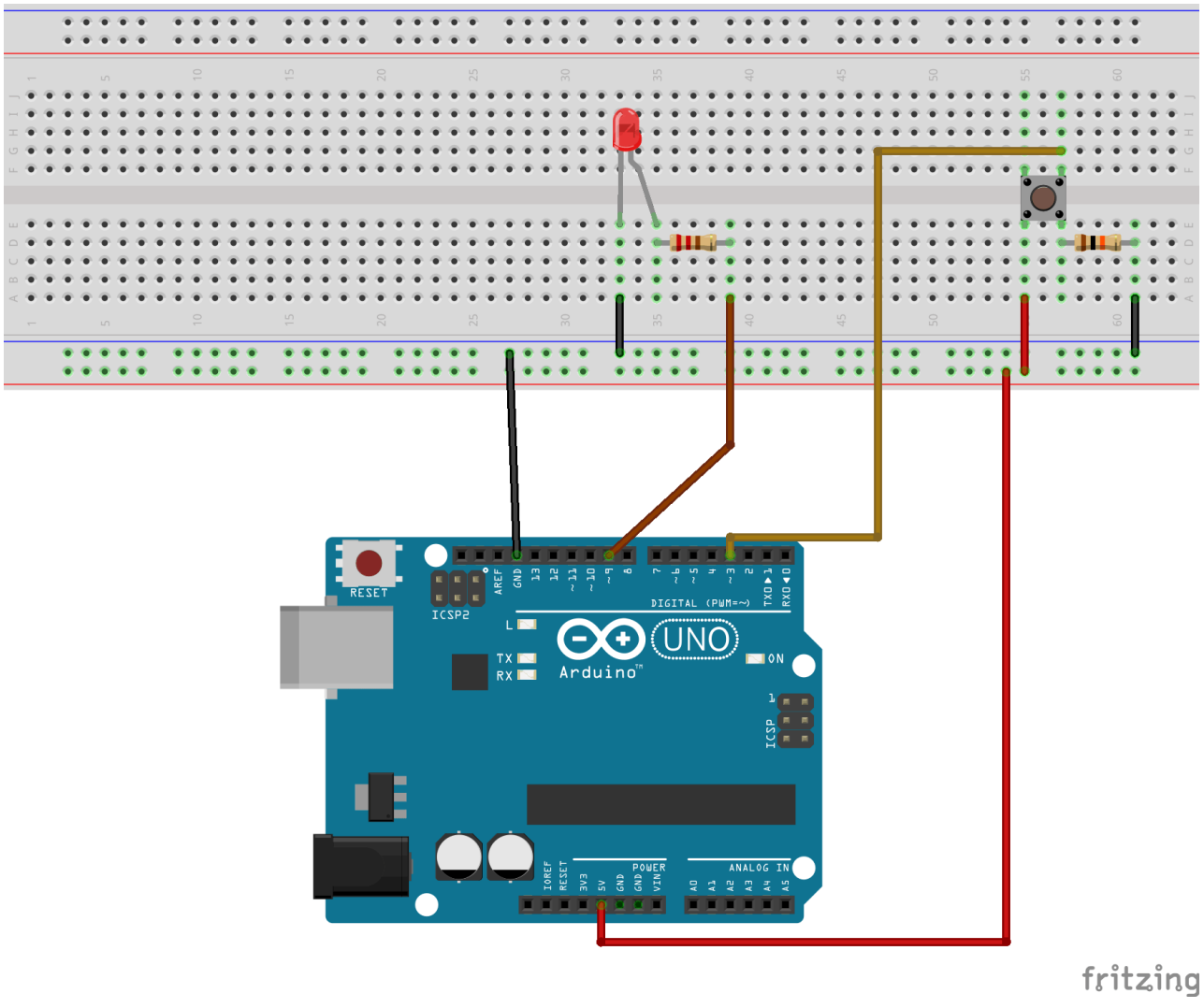
On this tutorial we'll be using an Arduino Uno board, so we only have two choices! We can either use pin 2 or pin 3.

If you want to use more interrupts in your programs, you can switch to the Arduino Mega. This board is really pretty close from the Arduino Uno, with more pins. And if you need even more interrupts, choose something like the Arduino Due – pay attention though, the Due works with 3.3V, not 5V.

Arduino Interrupts – Code example

For this tutorial we'll use a basic example. The goal of the program is to change the state of a LED when the user presses a push button.

Schematics



Note that we are using the pin 3 for the button. As previously stated, on Arduino Uno you can only use pin 2 and 3 for interrupts. Pay attention when you have to choose a pin for an interrupt. If the pin is not compatible with interrupts your program won't work (but still compile), and you'll spend quite some time scratching your head while trying to find a solution.

Types of interrupts

Arduino interrupts are triggered when there is a change in the digital signal you want to monitor. But you can choose exactly what you want to monitor. For that you'll have to modify the 3rd parameter of the attachInterrupt() function:

- **RISING**: Interrupt will be triggered when the signal goes from LOW to HIGH
- **FALLING**: Interrupt will be triggered when the signal goes from HIGH to LOW
- **CHANGE**: Interrupt will be triggered when the signal changes (LOW to HIGH or HIGH to LOW)
- **LOW**: Interrupt will be triggered whenever the signal is LOW

Practically speaking, you could monitor when the user presses the buttons, or when he/she releases the button, or both.

If you've added a pull-down resistor to the button – meaning its normal state is LOW – then monitoring when it's pressed means you have to use RISING. If you've added a pull-up resistor, the button state is already HIGH, and you have to use FALLING to monitor when it's pressed (linked to the ground).

Arduino code without interrupts

```
1. #define LED_PIN 9
2. #define BUTTON_PIN 3
```

```
3.
4.  byte ledState = LOW;
5.
6.  void setup() {
7.    pinMode(LED_PIN, OUTPUT);
8.    pinMode(BUTTON_PIN, INPUT);
9.  }
10.
11. void loop() {
12.   if (digitalRead(BUTTON_PIN), HIGH) {
13.     ledState = !ledState;
14.   }
15.
16.   digitalWrite(LED_PIN, ledState);
17. }
```

Nothing really new here. We initialize the pin of the LED as OUTPUT and the pin of the button as INPUT. In the loop() we monitor the button state and modify the LED state accordingly. Note that for simplicity I haven't use a debounce on the button.

Arduino code with interrupts

```
1.  #define LED_PIN 9
2.  #define BUTTON_PIN 3
3.
4.  volatile byte ledState = LOW;
5.
6.  void setup() {
7.    pinMode(LED_PIN, OUTPUT);
8.    pinMode(BUTTON_PIN, INPUT);
9.
10.   attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), blinkLed, RISING);
11. }
12.
13. void loop() {
14.   // nothing here!
15. }
16.
17. void blinkLed() {
18.   ledState = !ledState;
19.   digitalWrite(LED_PIN, ledState);
20. }
```

Here we changed the way we are monitoring the push button. Instead of polling its state, there is now an interrupt function attached to the pin. When the signal on the button pin is rising – which means it's going from LOW to HIGH, the current program execution – loop() function – will be stopped and the blinkLed() function will be called. Once blinkLed() has finished, the loop() can continue.

Here, the main advantage you get is that there is no more polling for the button in the loop() function. As soon as the button is pressed, blinkLed() will be called, and you don't need to worry about it in the loop().

As you might have noticed, we use the keyword “volatile” in front of the ledState variable. I’ll explain you later in this post why we need that.

You have to use the attachInterrupt() function to attach a function to an interrupt pin. This function takes 3 parameters: the interrupt pin, the function to call, and the type of interrupt.

Five things you need to know about Arduino Interrupts

Keep the interrupts fast

As you can guess, you should make the interrupt function as fast as possible, because it stops the main execution of your program. You can’t do heavy computation. Also, only one interrupt can be handled at a time.

What I recommend you to do is to only change state variables inside interrupt functions. In the main loop(), you check for those state variables and do any required computation or action.

Let's say you want to move a motor, and this action is triggered by an interrupt. In this case, you could have a variable named "shouldMoveMotor" that you set to "true" in the interrupt function.

In your main program, you check for the state of the "shouldMoveMotor". When it's true, you start moving the motor.

```
#define BUTTON_PIN 3

volatile bool shouldMoveMotor = false;

void setup() {
  pinMode(BUTTON_PIN, INPUT);
  attachInterrupt(digitalPinToInterrupt(BUTTON_PIN), triggerMoveMotor, RISING);
}

void loop() {
  if (shouldMoveMotor) {
    shouldMoveMotor = false;
    moveMotor();
  }
}

void triggerMoveMotor() {
  shouldMoveMotor = true;
}

void moveMotor() {
  // this function may contains code that
  // requires heavy computation, or takes
  // a long time to execute
}
```

And you can do exactly the same for a heavy computation, for example if the computation takes more than a few microseconds to complete.

If you don't keep the interrupts fast, you might miss important deadlines in your code. For a mobile robot with 2 wheels, that may make the motor movement jerky. For communication between devices, you might miss some data, etc.

When you need to deal with real-time constraints, this rule becomes even more important.

Time functionalities and interrupts

A basic rule of thumb: don't use time functionalities in your interrupts. Here's more details about the 4 main time functions:

- **millis()**: this will return the time spent since the Arduino program has

started, in milliseconds. This function relies on some other interrupts to count, and as you are inside an interrupt, other interrupts are not running. Thus, if you use millis(), you'll get the last stored value, which will be correct, but when inside the interrupt function, the millis() value will never increase.
- **delay()**: this one will simply not work, as it also relies on interrupts. Plus, even if it was possible, you should not use it because you now know that you have to keep the interrupts very fast.
- **micros()**: this function is the same as millis(), but returns the time in microseconds. However, contrary to millis(), micros() will work at the beginning of an interrupt. But after 1 or 2 milliseconds, the behavior won't be accurate and you may have a permanent drift every time you use micros() afterwards. Again, the advice is the same: make your interrupts short and fast!
- **delayMicroseconds()**: this one will work as usual, but… Don't use it. As you saw before, there are too many things that can go wrong if you stay too long in an interrupt.

All in all, you should avoid using those functions.

Maybe using millis() or micros() can sometimes be useful, if you want to make a comparison of duration (for example to debounce a button). But you can also do that in your code, using the interrupt only to notify of a change in the state of the monitored signal.

Don't use the Serial library inside interrupts

The Serial library is very useful to debug and communicate between your Arduino board and another board or device. But it's not a great fit for interrupt functions.

When you are inside an interrupt, the received Serial data may be lost. Thus it's not a good idea to use the reading functionalities of Serial. Also if you make the interrupt too long, and read from Serial after that in your main code, you may still have lost some parts of the data.

You can use Serial.print() inside an interrupt for debugging, for example if you're not sure when the interrupt is triggered. But it also has its own source of problems.

The best way to print something from an interrupt, is simply to set a flag inside the interrupt, and poll this flag inside the main loop() program. When the flag is turned

on, you print something, and turn off the flag. Doing that will save you from potential headaches.

Volatile variables

If you modify a variable inside an interrupt, then you should declare this variable as volatile.

The compiler does many things to optimize the code and the speed of the program. This is a good thing, but here we need to tell it to “slow down” on optimization.

For example, if the compiler sees a variable declaration, but the variable is not used anywhere in the code (except from interrupts), it may remove that variable. With a volatile variable you’re sure that it won’t happen, the variable will be stored anyway.

Also, when you use volatile it tells the controller to reload the variable whenever it’s referenced. Sometimes the compiler will use copies of variables to go faster. Here you want to make sure that every time you access/modify the variable, either in the main program or inside an interrupt, you get the real variable and not a copy.

Note that only variables that are used inside and outside an interrupt should be declared as volatile. You don’t want to unnecessarily slow down your code.

Interrupts parameters and returned value

An interrupt function can’t take any parameter, and it doesn’t return any value. Basically if you had to write a prototype for an interrupt this would be something like `void interruptFunction();`.

Thus, the only way to share data with the main program is through global volatile variables. In an interrupt you can also get and set data from hardware pins, as long as you keep the program short. For example, using digitalRead() or digitalWrite() may be OK if you don’t abuse it.

Conclusion

Arduino interrupts are very useful when you want to make sure you don’t miss any change in a signal you monitor (on a digital pin mostly).

However, during this post you saw that there are many rules and limitations when using interrupts. This is something you should handle with care, and not use too much. Sometimes, using simple polling may be more appropriate, if for example you manage to write an efficient and deterministic multitasking Arduino program.

Interrupts can also be used just to trigger a flag, and you keep using the polling technique inside your main loop() – but this time, instead of monitoring the hardware pin, you monitor the software flag.

The main takeaway for you, if you want to use interrupts in your code: keep your interrupts short. Thus you will avoid many unnecessary and hard-to-debug problems.

Arduino Tutorials