Multi-tasking the Arduino - Part 2

Created by Bill Earl

https://learn.adafruit.com/multi-tasking-the-arduino-part-2

Last updated on 2024-03-08 02:01:37 PM EST

Table of Contents

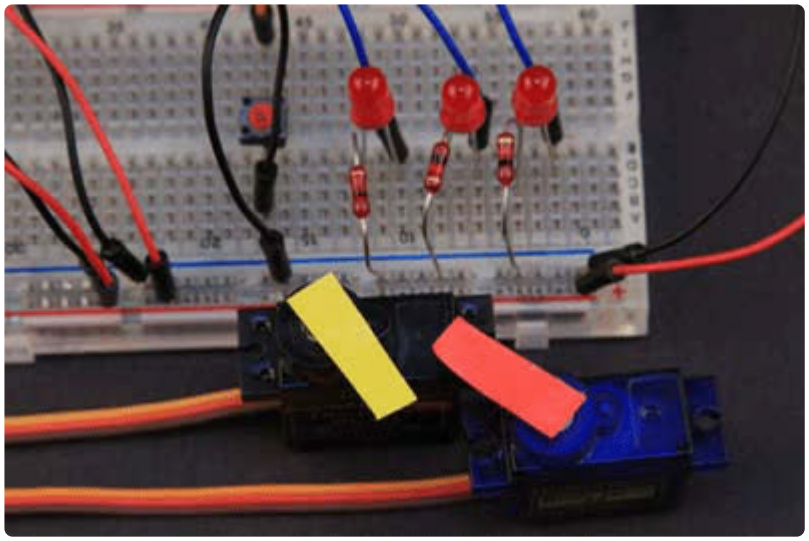
Overview

Often in a project you need the processor to do just one more thing. How can you make it do that when it is busy with other things?

One Man Band photo circa 1865 by Knox via Wikimedia is in the public domain

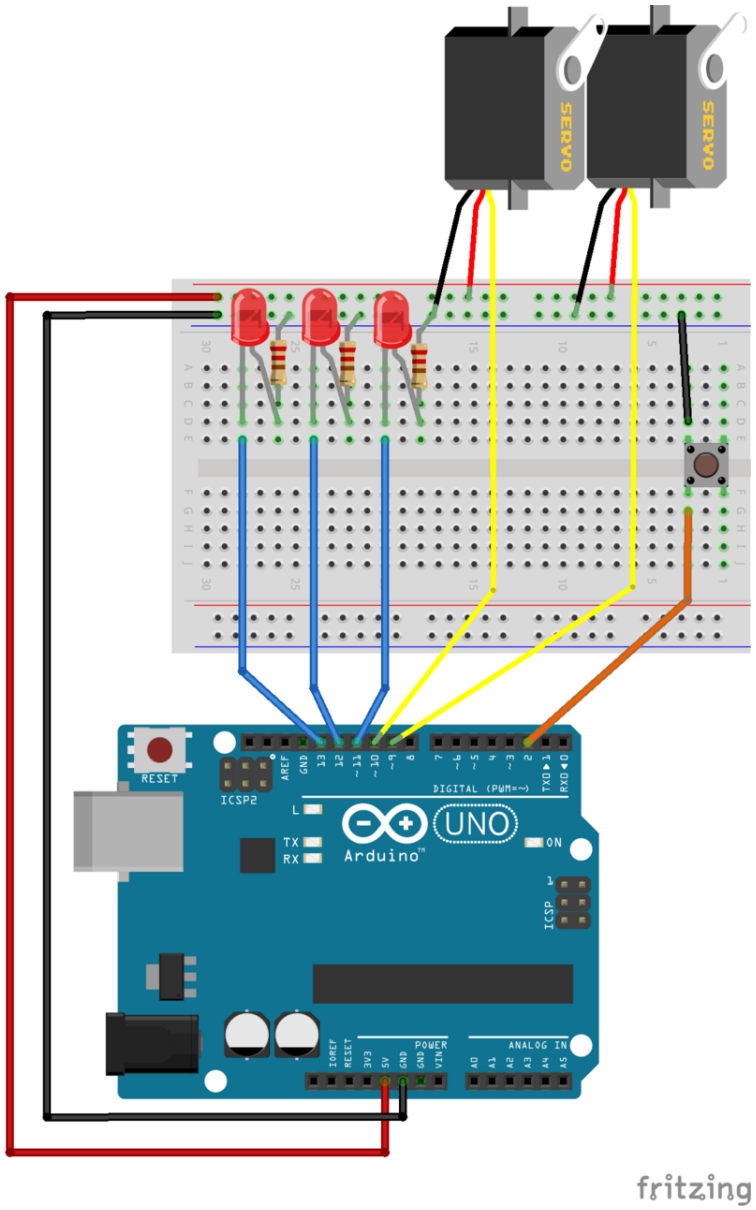
In this guide we'll build on the techniques learned in Part 1 (https://adafru.it/mEf) of the series while we explore several types of Arduino interrupts and show how they can be used to help your Arduino juggle even more tasks while keeping your code simple and responsive.

We'll learn how to harness the timer interrupts to keep everything running like clockwork. And we'll explore using external interrupts to give us notifications of external events.



Setup

For all the examples in this guide, the following wiring will be used:



What is an Interrupt?

An interrupt is a signal that tells the processor to immediately stop what it is doing and handle some high priority processing. That high priority processing is called an Interrupt Handler.

An interrupt handler is like any other void function. If you write one and attach it to an interrupt, it will get called whenever that interrupt signal is triggered. When you return from the interrupt handler, the processor goes back to continue what it was doing before.

Where do they come from?

Interrupts can be generated from several sources:

- Timer interrupts from one of the Arduino timers.
- External Interrupts from a change in state of one of the external interrupt pins.
- Pin-change interrupts from a change in state of any one of a group of pins.

What are they good for?

Using interrupts, you don't need to write loop code to continuously check for the high priority interrupt condition. You don't have to worry about sluggish response or missed button presses due to long-running subroutines.

The processor will automagically stop whatever it is doing when the interrupt occurs and call your interrupt handler. You just need to write code to respond to the interrupt whenever it happens.

Timer Interrupts

Don't call us, we'll call you

In part 1 of this series, we learned how to use millis() for timing. But in order to make that work, we had to call millis() every time through the loop to see if it was time to do something. It is kind of a waste to be calling millis() more than once a millisecond,

only to find out that the time hasn't changed. Wouldn't it be nice if we only had to check once per millisecond?

Timers and timer interrupts let us do exactly that. We can set up a timer to interrupt us once per millisecond. The timer will actually call us to let us know it is time to check the clock!

Arduino Timers

The Arduino Uno has 3 timers: Timer0, Timer1 and Timer2. Timer0 is already set up to generate a millisecond interrupt to update the millisecond counter reported by millis(). Since that is what we are looking for, we'll get Timer0 to generate an interrupt for us too!

Frequency and Counts

Timers are simple counters that count at some frequency derived from the 16MHz system clock. You can configure the clock divisor to alter the frequency and various different counting modes. You can also configure them to generate interrupts when the timer reaches a specific count.

Timer0 is an 8-bit that counts from 0 to 255 and generates an interrupt whenever it overflows. It uses a clock divisor of 64 by default to give us an interrupt rate of 976.5625 Hz (close enough to a 1KHz for our purposes). We won't mess with the freqency of Timer0, because that would break millis()!

Comparison Registers

Arduino timers have a number of configuration registers. These can be read or written to using special symbols defined in the Arduino IDE. For comprehensive description of all these registers and their functions, see the links in "**For further reading**" below.

We'll set up a up a **comparison register** for Timer 0 (this register is known as OCR0A) to generate another interrupt somewhere in the middle of that count. On every tick, the timer counter is compared with the comparison register and when they are equal an interrupt will be generated.

The code below will generate a 'TIMER0_COMPA' interrupt whenever the counter value passes 0xAF.

```
  // Timer0 is already used for millis() - we'll just interrupt somewhere
  // in the middle and call the "Compare A" function below
  OCR0A = 0xAF;
  TIMSK0 |= _BV(OCIE0A);
```

Then we'll define an interrupt handler for the timer interrupt vector known as "TIMER0_COMPA_vect". In this interrupt handler we'll do all the stuff we used to do in the loop.

```
// Interrupt is called once a millisecond,
SIGNAL(TIMER0_COMPA_vect)
{
  unsigned long currentMillis = millis();
  sweeper1.Update(currentMillis);

  //if(digitalRead(2) == HIGH)
  {
     sweeper2.Update(currentMillis);
     led1.Update(currentMillis);
  }

  led2.Update(currentMillis);
  led3.Update(currentMillis);
}
```

Which leaves us with a completely empty loop.

```
void loop()
{
}
```

You can do anything you want in your loop now. You can even be decadent and use a delay()! The flashers and sweepers won't care. They will still get called once per millisecond regardless!

For further reading:

This is just a simple example of what timers can do. For more in-depth information on the different types of timers and ways to configure them, check out the "Libraries and Links" page.

The Source Code:

Here's the whole code, including the flashers and sweepers:

```
#include &lt;Servo.h&gt;

class Flasher
{
    // Class Member Variables
```

```
    // These are initialized at startup
    int ledPin;      // the number of the LED pin
    long OnTime;     // milliseconds of on-time
    long OffTime;    // milliseconds of off-time

    // These maintain the current state
    int ledState;                 // ledState used to set the LED
    unsigned long previousMillis;   // will store last time LED was updated

  // Constructor - creates a Flasher
  // and initializes the member variables and state
  public:
  Flasher(int pin, long on, long off)
  {
    ledPin = pin;
    pinMode(ledPin, OUTPUT);

    OnTime = on;
    OffTime = off;

    ledState = LOW;
    previousMillis = 0;
  }

  void Update(unsigned long currentMillis)
  {
    if((ledState == HIGH) &amp;&amp; (currentMillis - previousMillis &gt;= OnTime))
    {
    ledState = LOW;  // Turn it off
      previousMillis = currentMillis;  // Remember the time
      digitalWrite(ledPin, ledState);  // Update the actual LED
    }
    else if ((ledState == LOW) &amp;&amp; (currentMillis - previousMillis &gt;=
OffTime))
    {
      ledState = HIGH;  // turn it on
      previousMillis = currentMillis;   // Remember the time
      digitalWrite(ledPin, ledState);     // Update the actual LED
    }
  }
};

class Sweeper
{
  Servo servo;              // the servo
  int pos;              // current servo position
  int increment;        // increment to move for each interval
  int  updateInterval;      // interval between updates
  unsigned long lastUpdate; // last update of position

public:
  Sweeper(int interval)
  {
    updateInterval = interval;
    increment = 1;
  }

  void Attach(int pin)
  {
    servo.attach(pin);
  }

  void Detach()
  {
    servo.detach();
  }

  void Update(unsigned long currentMillis)
  {
```

```
    if((currentMillis - lastUpdate) &gt; updateInterval)  // time to update
    {
      lastUpdate = millis();
      pos += increment;
      servo.write(pos);
      if ((pos &gt;= 180) || (pos &lt;= 0)) // end of sweep
      {
        // reverse direction
        increment = -increment;
      }
    }
  }
};


Flasher led1(11, 123, 400);
Flasher led2(12, 350, 350);
Flasher led3(13, 200, 222);

Sweeper sweeper1(25);
Sweeper sweeper2(35);

void setup()
{
  sweeper1.Attach(9);
  sweeper2.Attach(10);

  // Timer0 is already used for millis() - we'll just interrupt somewhere
  // in the middle and call the "Compare A" function below
  OCR0A = 0xAF;
  TIMSK0 |= _BV(OCIE0A);
}

// Interrupt is called once a millisecond, to update the LEDs
// Sweeper2 s not updated if the button on digital 2 is pressed.
SIGNAL(TIMER0_COMPA_vect)
{
  unsigned long currentMillis = millis();
  sweeper1.Update(currentMillis);

  if(digitalRead(2) == HIGH)
  {
     sweeper2.Update(currentMillis);
     led1.Update(currentMillis);
  }

  led2.Update(currentMillis);
  led3.Update(currentMillis);
}

void loop()
{
}
```

External Interrupts

When it's better to be out of the loop

Unlike timer interrupts, external interrupts are triggered by external events. For example, when a button is pushed or you receive a pulse from a rotary encoder.

However, just like the timer interrupts, you don't need to keep polling the GPIO pins for a change.

The Arduino UNO has 2 external interrupt pins. In this example, we'll attach our pushbutton to one of them and use it to reset our sweepers. First, well add a "reset()" function to our sweeper class. The reset() function sets the position to 0 and immediately positions the servo there:

```
  void reset()
  {
    pos = 0;
    servo.write(pos);
    increment = abs(increment);
  }
```

Next, we'll add a call to AttachInterrupt() (https://adafru.it/dd4) to connect the external interrupt with our handler code.

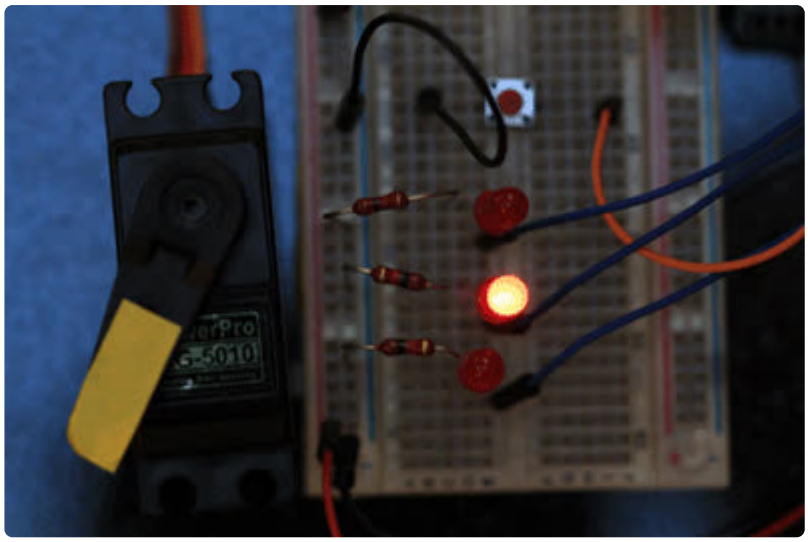
On the UNO, Interrupt 0 is associated with digital pin 2. We tell it to look for the "FALLING" edge of the signal on that pin. When the button is pressed, the signal "falls" from HIGH to LOW and the "Reset" interrupt handler is called.

```
  pinMode(2, INPUT_PULLUP);
  attachInterrupt(0, Reset, FALLING);
```

And here is the "Reset" Interrupt Handler. It just calls the sweeper reset functions:

```
void Reset()
{
  sweeper1.reset();
  sweeper2.reset();
}
```

Now, whenever you press the button, the servos stop what they are doing and immediatey seek to the zero position.



The Source Code:

Here's the complete sketch with timers and external interrupts:

```
#include &lt;Servo.h&gt;

class Flasher
{
    // Class Member Variables
    // These are initialized at startup
    int ledPin;      // the number of the LED pin
    long OnTime;     // milliseconds of on-time
    long OffTime;    // milliseconds of off-time

    // These maintain the current state
    volatile int ledState;                  // ledState used to set the LED
    volatile unsigned long previousMillis;  // will store last time LED was updated

  // Constructor - creates a Flasher
  // and initializes the member variables and state
  public:
  Flasher(int pin, long on, long off)
  {
    ledPin = pin;
    pinMode(ledPin, OUTPUT);

    OnTime = on;
    OffTime = off;

    ledState = LOW;
    previousMillis = 0;
  }

  void Update(unsigned long currentMillis)
  {
    if((ledState == HIGH) &amp;&amp; (currentMillis - previousMillis &gt;= OnTime))
    {
    ledState = LOW;  // Turn it off
      previousMillis = currentMillis;  // Remember the time
      digitalWrite(ledPin, ledState);  // Update the actual LED
    }
    else if ((ledState == LOW) &amp;&amp; (currentMillis - previousMillis &gt;=
OffTime))
    {
      ledState = HIGH;  // turn it on
```

```
      previousMillis = currentMillis;   // Remember the time
      digitalWrite(ledPin, ledState);     // Update the actual LED
    }
  }
};

class Sweeper
{
  Servo servo;              // the servo
  int  updateInterval;      // interval between updates

  volatile int pos;                  // current servo position
  volatile unsigned long lastUpdate; // last update of position
  volatile int increment;            // increment to move for each interval

public:
  Sweeper(int interval)
  {
    updateInterval = interval;
    increment = 1;
  }

  void Attach(int pin)
  {
    servo.attach(pin);
  }

  void Detach()
  {
    servo.detach();
  }

  void reset()
  {
    pos = 0;
    servo.write(pos);
    increment = abs(increment);
  }

  void Update(unsigned long currentMillis)
  {
    if((currentMillis - lastUpdate) &gt; updateInterval)  // time to update
    {
      lastUpdate = currentMillis;
      pos += increment;
      servo.write(pos);
      if ((pos &gt;= 180) || (pos &lt;= 0)) // end of sweep
      {
        // reverse direction
        increment = -increment;
      }
    }
  }
};


Flasher led1(11, 123, 400);
Flasher led2(12, 350, 350);
Flasher led3(13, 200, 222);

Sweeper sweeper1(25);
Sweeper sweeper2(35);

void setup()
{
  sweeper1.Attach(9);
  sweeper2.Attach(10);

  // Timer0 is already used for millis() - we'll just interrupt somewhere
```

```
  // in the middle and call the "Compare A" function below
  OCR0A = 0xAF;
  TIMSK0 |= _BV(OCIE0A);

  pinMode(2, INPUT_PULLUP);
  attachInterrupt(0, Reset, FALLING);
}

void Reset()
{
  sweeper1.reset();
  sweeper2.reset();
}

// Interrupt is called once a millisecond,
SIGNAL(TIMER0_COMPA_vect)
{
  unsigned long currentMillis = millis();
  sweeper1.Update(currentMillis);

  //if(digitalRead(2) == HIGH)
  {
     sweeper2.Update(currentMillis);
     led1.Update(currentMillis);
  }

  led2.Update(currentMillis);
  led3.Update(currentMillis);
}

void loop()
{
}
```

Libraries and Links

More About Timers

Timers can be configured to run at various frequencies and operate in different modes. In addition to generating interrupts, they are also used to control the PWM pins. The links below are excellent resources for understanding how to configure and use timers:

Secrets of Arduino PWM

https://adafru.it/edS

Timer/PWM Cheat Sheet

https://adafru.it/edT

Timer Libraries

There are a number of Arduino 'timer' libraries available on the web. Many simply monitor millis() and require constant polling as we did in part 1 of this series. But there are a few that actually let you configure timers to generate interrupts.

Paul Stoffregan's excellent TimerOne and TimerThree libraries take care of many of the low-level details of timer interrupt configuration. (Note that TimerThree is not applicable to the UNO. It can be used with the Leonardo, Mega and some of the Teensy boards)

TimerOne and Timer Three Libraries

https://adafru.it/edU

Pin Change Interrupts

For when 2 is not enough

The Arduino UNO has only 2 external interrupt pins. But what if you need more than 2 interrupts? Fortunately, the Arduino UNO supports “pin change” interrupts on all pins.

Pin change interrupts are similar to external interrupts. The difference is that one interrupt is generated for a change in state on any of the 8 associated pins. These are a little more complex to handle, since you have to track the last known state of all 8 pins to figure out which of the 8 pins caused the interrupt.

The PinChangeInt library at the Arduino Playground implements a handy interface for pin change interrupts: http://playground.arduino.cc/Main/PinChangeInt

PinChangeInt Library

https://adafru.it/edV

Timer and Interrupt Etiquette

Interrupts are like the express lane at the supermarket. Be considerate and keep it to 10 items or less and everything will run smoothly.

If everything is high prority, then nothing is high priority.

Interrupts handlers should be used for processing high-priority, time-sensitive events only. Remember that interrupts are disabled while you are in the interrupt handler. If you try to do too much at the interrupt level, you will degrade response to other interrupts.

One interrupt at a time.

When in the ISR, interrupts are disabled. This has two very important implications:

1. Work done in the ISR should be kept short so as not to miss any interrupts.
2. Code in the ISR should not call anything that requires interrupts to be active (e.g. delay() or anything that uses the i2c bus). This will result in hanging your program.

Defer lengthy processing to the loop.

If you need to do extensive processing in response to an interrupt, use the interrupt handler to do only what is essential, then set a volatile state variable (see below) to indicate that further processing is required. When you call your update function from the loop, check the state variable to see if any follow-up processing is required.

Check before re-configuring a timer

Timers are a limited resource. There are only 3 on an UNO and they are used for many things. If you mess with a timer configuration, some other things may not work anymore. For example, on an Arduino UNO:

- **Timer0** - used for millis(), micros(), delay() and PWM on pins 5 & 6
- **Timer1** - used for Servos, the WaveHC library and PWM on pins 9 & 10
- **Timer2** - used by Tone and PWM on pins 3 & 11

Share Data Safely

Because an interrupt will suspend whatever the processor is doing to process the interrupt, we have to be careful about sharing data between interrupt handlers and the code in our loop.

Volatile Variables

Sometimes the compiler will try to optimize your code for speed. Sometimes these optimizations will keep a copy commonly used variables in a register for fast access. The problem is, if one of those variables is shared between the interrupt handler and the loop code, one of them may end up looking at a stale copy instead of the real thing. Marking the variable as voltatile tells the compiler not to do those potentially dangerous kids of optimizations.

Protecting Larger Variables

Evan marking a variable **volatile** is not enough if it the variable is larger than an integer (e.g. strings, arrays, structures etc.). Larger variables require several instruction cycles to update, and if an interrupt occurs in the middle of that update, the data can be corrupted. If you have larger variables or structures that are shared with interrupt handlers, you should disable interrupts (https://adafru.it/edW) when updating them from the loop. (Interrupts are disabled in the interrupt handler already by default.)