Intermediate Microcontroller Tutorial

This tutorial is intended to explore more advanced techniques in microcontrollers to supplement and extend material covered in 2.678 and 2.007. It is assumed you have taken one of those classes already, or have significant other experience working with Arduinos.

In this project we’re going to build and program a simple device that cycles through different LED light patterns when you press a button. The device is simple in operation but its implementation involves interrupt programming, defining a new target device, and direct register programming of an ATTiny85 microcontroller.

1. Hello, World
   It doesn’t matter how many times you’ve done it — especially when you’re venturing into new terrain, you should
   (a) Plug in your Arduino Uno,
   (b) Make sure the drivers are installed,
   (c) Fire up Examples → Basics → Blink,
   (d) ...and make sure you can actually program the Arduino, at all, before you get started with something new.

   Fun fact! Arduinos come from the factory with Blink pre-installed. Make sure you change the blink rate to make absolutely sure the Arduino is blinking because you said so.

2. Set up the hardware for Arduino
   In general, it’s a good idea to build the smallest possible system to test a new design. We’re therefore going to build and test the system first on a standard Arduino Uno, then port it to our desired target.

   Use a breadboard and jumper wires to build the following circuit using an RGB LED and one of our small tactile switches:
3. Back to Coding

Again, we proceed in baby steps, to avoid an impossible debugging situation later. To start, modify your `BLINK` code to turn on each of the three LEDs, thus testing whether they’re wired correctly (this could be three different programs — your goal here is speed, not elegance). Then, write a simple switch test — for example:

```c
void setup()
{
  pinMode(5, OUTPUT);
  pinMode(2, INPUT_PULLUP);
}

void loop()
{
  if (digitalRead(2) == HIGH)
    digitalWrite(5, HIGH);
  else
    digitalWrite(5, LOW);
}
```

Upload this to the Uno to confirm the switch is working as expected.

**Finally** (now that we know the hardware works), we can get to work on the actual project at hand. We want to design a device that

(a) Fades an LED,

(b) Constantly watches for button presses, and

(c) Changes which LED is fading (immediately!) every time you press the button.
We’re going to do this by modifying the *Fade* example code. Start by opening up *Examples → Basics → Fade*, and taking a brief look at it to remind yourself why it works (you can upload it to your Uno, too — it should fade the LED connected to pin 9).

Now we’re going to make a few alterations, which we’ll explain after getting them into the code. Modify the *Fade* code to include the (numerous) additions below:

```c
volatile int state = 0;
volatile unsigned long last_interrupt_time = 0;
int ledPin = 5;

void setup() {
  pinMode(5, OUTPUT);
  pinMode(6, OUTPUT);
  pinMode(9, OUTPUT);
  pinMode(2, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(2), counter_ISR, FALLING);
}

void loop() {
  if (state == 0) ledPin = 5;
  if (state == 1) ledPin = 6;
  if (state == 2) ledPin = 9;

  for (int fadeValue = 0; fadeValue <= 255; fadeValue += 5) {
    if (state == 0) ledPin = 5;
    if (state == 1) ledPin = 6;
    if (state == 2) ledPin = 9;
    analogWrite(ledPin, fadeValue);
    delay(30);
  }

  for (int fadeValue = 255; fadeValue >= 0; fadeValue -= 5) {
    if (state == 0) ledPin = 5;
    if (state == 1) ledPin = 6;
    if (state == 2) ledPin = 9;
    analogWrite(ledPin, fadeValue);
    delay(30);
  }
}
```
void counter_ISR() {
    unsigned long interrupt_time = millis();

    if (interrupt_time - last_interrupt_time > 200)
    {
        state++;
        if (state > 2) state = 0;
        digitalWrite(ledPin, LOW);
        last_interrupt_time = interrupt_time;
    }
}

This code probably includes a few new things:

(a) The volatile keyword lets the compiler know that this variable can change without being modified by the main code. Without this keyword, the compiler might analyze your code, notice that the variable state is never changed within loop(), and optimize the code by removing all of your if statements — because if state is a constant, why check it?

(b) attachInterrupt() creates a link between some interrupt event and a function in your code. Whenever the event happens, a few new things happen:

i. Code execution halts, wherever you happen to be in loop().

ii. That location is saved, to be returned to later.

iii. Interrupt events are turned off (so you can’t interrupt an interrupt).

iv. The microcontroller then jumps to the appropriate interrupt service routine (ISR) that was linked to the interrupt event. There are many possible sources of an interrupt, each of which can trigger a different chunk of code.

v. After the ISR is done, code execution resumes at the saved location and interrupts are turned back on.

There are a number of behind-the-scenes operations that are interrupt-driven. The most famous of these (to 2.678 alumni) is the millis() function, which stores elapsed time in a variable that is incremented by a timer interrupt. Because an active interrupt disables all other, possibly mission-critical, interrupts, it is usually desirable to write ISR functions that are as fast as possible, and offload as much computation to regular operation in loop().

In this specific case, our attachInterrupt function looks to digital pin 2 for a falling digital voltage. When that falling edge occurs, the counter_ISR() function is triggered.
(c) A simple state machine: Our `loop()` function is written so that the details of its operation hinge on a single variable formally called the state variable. We have built a very simple implementation of it here but forking the behavior of a function based on state is generally an extremely powerful way to react to interrupting events. Note that because we’re working with Fade, and have for loops within `loop()`, we need to check state several times if we want an instant response. (There is a way to re-write `loop()` to avoid this problem, but we didn’t want to change it too much from the example code.)

(d) Switch “debouncing” – switches are noisy, and microcontrollers are almost inconceivably fast. When you close the switch, its two contacts “bounce” against each other like the world’s fastest snare drum roll. This is completely invisible to you. A microcontroller, however, will see the first contact, activate the ISR, complete it, and be ready to react to the next bounce. The timing code in our ISR makes it so that the ISR can only be triggered once every 200 milliseconds, so it will react to the first switch bounce and ignore the rest.

Compile the code and send it to the Uno. Now, you should be able to change the fade color by pressing the button.

4. A Minimal Target

Now we are going to port our code to the ATTiny85, an 8-pin microcontroller that in addition to being much smaller sells for less than $1 in small quantities. Its pinout is shown below:

Note the many possible functions for every pin that’s not VCC or GND.

First, let’s wire up the ATTiny85. Remove all the wires between the Uno and your breadboard, then modify the breadboard so that you have:
You should be able to build this by leaving your LED, resistors, and switch in the breadboard, adding the ATTiny85, and adding jumper wires to make the appropriate new connections to the new microcontroller. For now, connect VCC to the red column and GND to the blue column – you will be using the Uno to supply those with power in a little bit.

There are a number of “standard” Arduino components missing from this design. We’re not going to miss them, but you should know that we have left out:

(a) A serial communications channel — we do not have `Serial.print()` available to us, because I/O pins are too precious.

(b) An external oscillator — most AVR microcontrollers can self-oscillate up to a certain frequency. To run them at their maximum speed, or minimum voltage, you need an external resonator to keep them stable (see if you can find it on the Uno – it’s 16 MHz). We are doing neither of these things, so we’re going to run the device using its internal 8 MHz oscillator and save a part.

(c) A power source — normally you need a regulator (like an LM7805) to provide a stable power supply for the microcontroller. For today, we’re going to cheat and use the Uno as a power source, piggybacking on its regulated supply.

5. Arduino as ISP

All devices sold as “Arduino” come from the factory with code pre-installed. This code, called a bootloader, runs every time the Arduino is reset and follows this routine:

- **Wait ~1 second:** is the USB port trying to initiate programming?
  - **Yes:** Get code from USB port, write code to program storage, reset
  - **No:** Jump to program storage, run code starting there
The presence of this bootloader is what enables USB programming, but in exchange for its convenience we must pay two major penalties: First, our code does not run immediately upon reset because the bootloader must wait for programming every time. Second, we are obligated to connect an external communications channel to talk to the bootloader.

Our target chip does not have any of these extra communications channels, so we need to program it as a bare chip, with an In-System Programmer (ISP). This is a specialized device that can program a microcontroller directly, without a bootloader, while it is already installed in another system. These programming devices typically sell for about $50 — not having to buy one was one of the original selling points of the Arduino system itself.

Lucky for you, you can turn an Uno into an ISP! In the Arduino IDE, head to File → Examples → ArduinoISP. Make sure the Uno is disconnected from everything else (except the USB cable), and download this sketch to the Uno.

Now, use jumper wires to connect the following pins on the Uno and ATTiny85:

<table>
<thead>
<tr>
<th>Uno</th>
<th>ATTiny85</th>
</tr>
</thead>
<tbody>
<tr>
<td>5v</td>
<td>VCC (red column)</td>
</tr>
<tr>
<td>GND</td>
<td>GND (blue column)</td>
</tr>
<tr>
<td>D13</td>
<td>pin 7 (PB2)</td>
</tr>
<tr>
<td>D12</td>
<td>pin 6 (PB1)</td>
</tr>
<tr>
<td>D11</td>
<td>pin 5 (PB0)</td>
</tr>
<tr>
<td>D10</td>
<td>pin 1 (PB5)</td>
</tr>
</tbody>
</table>

For the record, you are connecting pins to use SPI (Serial Peripheral Interface) communication between the two microcontrollers. For example, the pin labeled MOSI (Master Out, Slave In) on the ATTiny85 is being connected to D11 which, it turns out, has an alternate function also named MOSI on the Uno. PB5 is the RESET pin for the ATTiny — resetting the chip is a part of the programming scheme.

In addition to this, on the Uno place a 1 \( \mu \)F capacitor between the header pins labeled RESET and GND.

6. **Hello, cheaper world!**

Now we can program the ATTiny! First, we need to teach the Arduino IDE what an ATTiny85 is.

(a) Go to File → Preferences and, in the “Additional Board Manager URLs” text box enter

```
https://raw.githubusercontent.com/damellis/attiny/ide-1.6.x-boards-manager/
package_damellis_attiny_index.json
```
(this, and only this, is available to you as an email by request so that you don’t have to worry about typing it in properly)

(b) Now go to **Tools → Board → Board Manager**, scroll down to the bottom, and install “attiny by David A. Mellis.”

(c) Finally, select **ATTiny** under **Tools → Board**, as well as **ATTiny85** for **Processor** and **8Mhz (internal)** for **Clock**.

(d) Under **Tools → Programmer**: select **Arduino as ISP** and then select **Burn Bootloader**. This burns the Arduino bootloader to the ATTiny85, which we don’t want, but at the same time also configures some extra parameters (called “fuses” for historical reasons), which we do need to modify. We’ll overwrite the bootloader in the next step.

(e) Moment of truth! Open up **Blink** again. Change the output pin to 0, 1 or 4 (these are pins PB0, PB1, or PB4). Make sure all your settings (Arduino as ISP, ATTiny85, 8Mhz) are still correct and then use **Sketch → Upload Using Programmer** to send the sketch to the ATTiny via the Uno. The LED should blink! Change the output pin to another color, upload (using programmer) again, and revel in your success.
7. WE DID IT

Finally, modify the `Fade` function from before as shown below:

```c
volatile int state = 0;
volatile unsigned long last_interrupt_time = 0;
int ledPin = 0;

void setup() {
  pinMode(0, OUTPUT);
  pinMode(1, OUTPUT);
  pinMode(4, OUTPUT);
  pinMode(2, INPUT_PULLUP);

  GIMSK |= _BV(INT0); // Enable INT0
  MCUCR &= 0b11111100; // Clear last two bits
  MCUCR |= 0b00000010; // Set last two bits
  sei();
}

void loop() {
  if (state == 0) ledPin = 0;
  if (state == 1) ledPin = 1;
  if (state == 2) ledPin = 4;

  digitalWrite(ledPin, HIGH); // Note we are blinking!
  delay(500);                 // (not fading)
  digitalWrite(ledPin, LOW);
  delay(500);
}

ISR(INT0_vect){
  unsigned long interrupt_time = millis();
  if (interrupt_time - last_interrupt_time > 200)
  {
    state++;
    if (state > 2) state = 0;
    digitalWrite(ledPin, LOW);
    last_interrupt_time = interrupt_time;
  }
}
```
Ooooooooy. There’s a lot of new stuff up there. You can upload this to your AT-Tiny85 and see that it works – the LED will blink, and the blink color will change when you press the button. But how? Most of the differences stem from the fact that the ATTiny is not as well-supported as the Uno, so you must talk to the microcontroller’s control registers directly.

(a) **GIMSK** — the behavior of the microcontroller is ultimately controlled by a very large number of memory registers which contain control settings. The General Interrupt Mask Register controls whether particular interrupt sources are active, as shown from the datasheet excerpt below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>GIMSK</td>
</tr>
</tbody>
</table>

- **9.3.2 GIMSK — General Interrupt Mask Register**

  - **Bits 7, 4:0 — Res: Reserved Bits**
    These bits are reserved bits in the ATTiny25/45/85 and will always read as zero.
  - **Bit 6 — INTO: External Interrupt Request 0 Enable**
    When the INTO bit is set (one) and the I-bit in the Status Register (SREG) is set (one), the external pin interrupt is enabled. The Interrupt Sense Control0 bits 1/0 (ISC01 and ISC00) in the MCU Control Register (MCUCR) define whether the external interrupt is activated on rising and/or falling edge of the INTO pin or level sensed. Activity on the pin will cause an interrupt request event if INTO is configured as an output. The corresponding interrupt of External Interrupt Request 0 is executed from the INTO Interrupt Vector.

(b) \( \text{||} = \_\text{BV} \text{(INT0)} \) — the \_\text{BV}() macro creates a byte with a single 1 set in a position — in this case, \text{INT0} = 6 and a 1 is placed into the 6th position of the byte. The \( \text{||} \) operator performs a bitwise Boolean OR between this byte and the existing contents of GIMSK, ultimately forcing the 6th bit to be a 1 and leaving the rest unchanged.

(c) **MCUCR** — the MCU Control Register tells the microcontroller what type of event to look for on the \text{INT0} pin, as detailed below:
We wish to direct the ATTiny to generate an interrupt event on “falling edge.” To accomplish that we must leave bits 2-7 unchanged while modifying bit 0 and bit 1. Two bitwise Boolean operations accomplish this without requiring us to know the pre-existing state of MCUCR — for example, 1 AND x always returns x, while 0 AND x always returns zero, so we can zero out bits 0 and 1 by performing \( \texttt{0b11111100} \ AND \ \texttt{MCUCR} \). (\texttt{0b} is the indicator to C that you’re using binary notation, just like \texttt{0x} signifies hexadecimal.)

(d) \texttt{sei()} — this is the directive to enable all interrupts.

All together, these four elements re-create the effect of \texttt{attachInterrupt()} from the Uno code. Again, because the ATTiny85 is not a fully-supported target for Arduino, we need to do some of these things from scratch, directly from the ATTiny datasheet.

That’s it! If you are somehow done early, come talk to Steve about how you might finish this code to enable the full fading routine. Because \texttt{analogWrite()} is also not fully supported, you are going to need to use more register programming (and the datasheet) to set up PWM output by hand as well.