# Button interrupts WITH debounce

**Goal**

The goal of this exercise is to learn how to add deboune protection to interrupts. In this part we'll but using some Bitwise operators and compound operators in our code. You may want to read about them on the reference page.

### Bitwise Operators

- & (bitwise and)
- | (bitwise or)
- ^ (bitwise xor)
- ~ (bitwise not)
- << (bitshift left)
- >> (bitshift right)

### Compound Operators

- ++ (increment)
- -- (decrement)
- += (compound addition)
- -= (compound subtraction)
- *= (compound multiplication)
- /= (compound division)
- &= (compound bitwise and)
- |= (compound bitwise or)

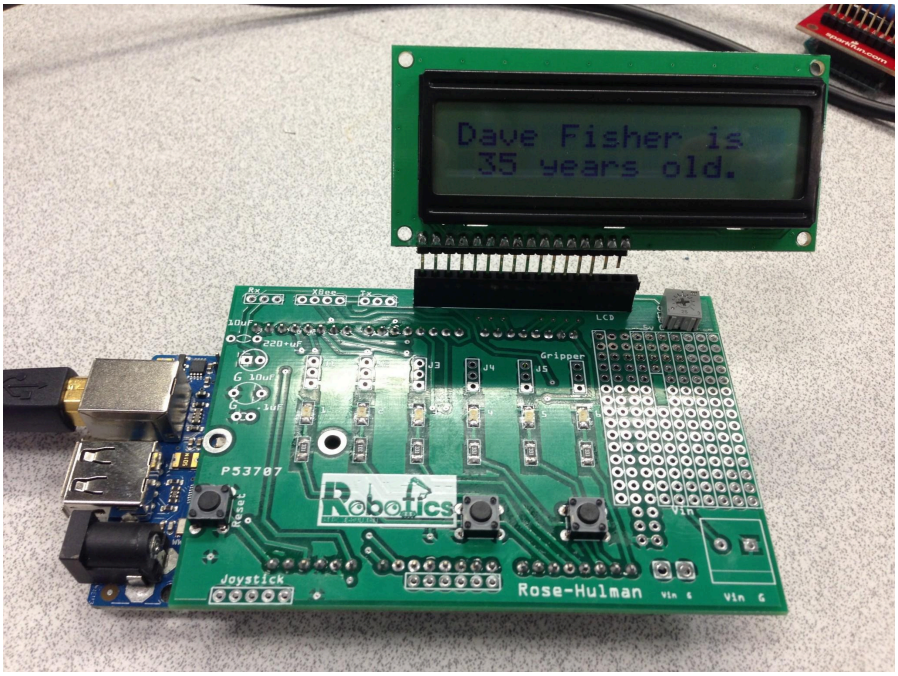
**Soldering**

None. You've already done the pushbuttons and LCD.

**Code**

We're going to build from the code in the last step to make sure you only age 1 year per 1 button press (instead of a few year per press).

Make sure your old code still runs:




The overall strategy of debounce is simple. Whenever you press or release a button it take about 20 mS for the signal to become stable. So whenever we see an edge we will set a flag (single bit of a variable is often used as a flag). Then 20 mS later we'll actually check the button to see if it's low (still pressed). If it is low then, and only then, will we change the age.

Make a variable called mainEventFlags. And make it a volatile int.

```
volatile int mainEventFlags = 0;
```

The word volatile is a hint to the compiler to NOT optimize code relating to this variable. For example if the compiler looked at this code:

```
mainEventFlags = 5;
while (mainEventFlags == 5) {
    mainEventFlags = 5;
}
doSomething();
```

It might decide that doSomething will never be called and remove it. However we DON'T want that because mainEventFlags might change in an interrupt. In general adding volatile should be necessary because the compiler will see it's used in the interrupt, but better safe than sorry.

Now let's talk about flags. There is no reason to use multiple variables for flags because it's either 0 or 1. So we'll use 1 variable (mainEventFlags) for 16 different flags. Eventually we'll used interrupts 0, 1, and 2 so we'll define a constant for each flag's bit.

```
#define FLAG_INTERRUPT_0 0x01
#define FLAG_INTERRUPT_1 0x02
#define FLAG_INTERRUPT_2 0x04
```

So Interrupt 0 uses the least significant bit of the mainEventFlags variable. Interrupt 1 uses the next bit, etc. So if

```
mainEventFlags = 0b0000000000000011;
```

Then both the Interrupt 0 and interrupt 1 flags are set. Here is code you can use to read and write individual flags

**Set a flag high**

```
mainEventFlags |= FLAG_INTERRUPT_0;
```

**Set a flag low**

```
mainEventFlags &= ~FLAG_INTERRUPT_0;
```

**Read a flag**

```
if (mainEventFlags & FLAG_INTERRUPT_0) {
  // do stuff
}
```

Bitwise operators aren't hard, but they may be new to you. If you didn't do it earlier, take a minute to review them on the Arduino reference page. ~ is the only one people find weird. There **is** a difference between Boolean operator ! and the Bitwise operator ~. Can you explain the difference? It's similar to the difference between & and && or | and ||. All are different.

### Bitwise Operators

- & (bitwise and)
- | (bitwise or)
- ^ (bitwise xor)
- ~ (bitwise not)
- << (bitshift left)
- >> (bitshift right)

The only one I ever mess up is resetting the flag to zero. Make sure that one makes sense to you. Combination of a bitwise not, bitwise and, and a compound operator. :)

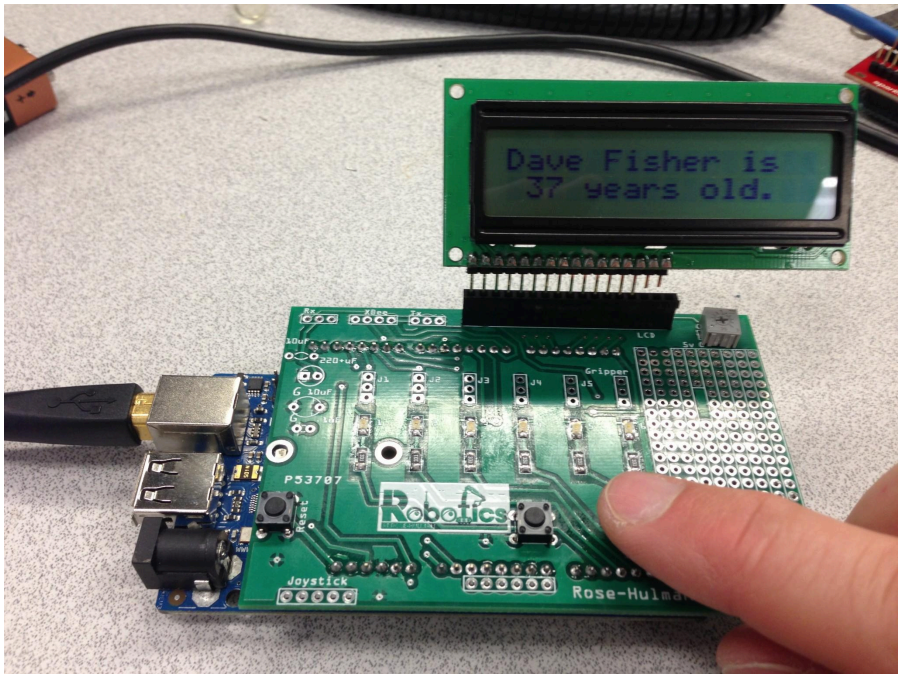
So the isr just got a lot simpler and the code within the loop will get harder. The ISRs will only set the flag (nothing more).

```
void int0_isr() {
  mainEventFlags |= FLAG_INTERRUPT_0;
}
void int1_isr() {
  mainEventFlags |= FLAG_INTERRUPT_1;
}
```

But then the loop function will need to check for flags.

```
if (mainEventFlags & FLAG_INTERRUPT_0) {
  delay(20);
  mainEventFlags &= ~FLAG_INTERRUPT_0;
  if (!digitalRead(PIN_RIGHT_BUTTON)) {
    age++;
  }
}
if (mainEventFlags & FLAG_INTERRUPT_1) {
  delay(20);
  mainEventFlags &= ~FLAG_INTERRUPT_1;
  if (!digitalRead(PIN_LEFT_BUTTON)) {
    age--;
  }
}
// Then LCD printing just like before
```

Does this code make sense? We are seeing if the flag is set. If it is then do nothing for 20 mS. After that reset the flag and check to see if the button is really pressed. Try it out. Just test. Press the button many times.




Do you get exactly 1 year per 1 press now? Hopefully it works slick.

Demo this part and get it checked off.

**Quick discussion about shortcomings (totally optional reading)**

This debounce method is simple and effective in most cases. However if the **loop** function is **long** this approach is too simple and makes a HORRIBLE assumption. The assumption is that the flags will be checked soonish ( < 250 ms). For example if you want to see the program you just made break, add this delay in to the loop.

```
delay(5000);
```

Ooops. That's deadly. :) No longer do you check the button state after 20mS. You only check the flags every 5 seconds and the 20mS check comes after the flag. Once you see the flag, sure it does the 20 mS thing and checks the button to still be low. However the button was pressed up to 5 seconds ago, so you might be checking the button 5.020 seconds later not .020 seconds later. A MUCH better solution is to start a timer when the interrupt happens to check the button in 20 mS from then regardless of the loop timing. Within a timer isr (fired 20mS after the press) check the button and set a flag there. The solution we used here WILL work in most of our programs since we'll try to keep the loop quick (< 250 mS), but long loop functions (> 500 mS) beware!

We will show you how to fix this probably in a later lab by using the TimerEventScheduler library

with a TimerEvent.  But to be honest that's usually more code than you need and overkill for sketches that have short loop functions.  If you want to see an example though, look in the Examples -> TimerEventScheduler for example code to fix this issue.  Not a concern right now.