# Buttons, events, and state machines

## Introduction

It sounds so easy: the button is either pressed or not. Why all the fuss?

In this lab you'll explore buttons and switches. You'll also explore the phenomenon of *bouncing* and learn how to combat it. In the process, you'll be introduced to event-driven programming, a very useful paradigm for organizing your codes. You'll finish by making a minimalist home alarm system to prove that you can implement many of the techniques demonstrated in class up to this point.

### Objectives

Upon successful completion of this lab, the student will be able to:

- Implement an event-driven program for reading a button,
- Implement button debouncing,
- Implement a state machine for reading a keypad, and
- Demonstrate the application of a number of concepts from the course as a whole.

## Preparation

### Background materials

Review the following materials:

**Switches and debouncing.** In Carryer *et al.*, read Section 13.4 up to and including 13.4.2.

**Event-driven programming and state machines.** See readings posted on collab.

## In lab

### Reading a button and debouncing

Here, you will implement an *event-driven* button counter, explore the bouncing phenomenon, and learn how to combat it. (Note that this was written before we went over some of this in class. You may, for example, already have a decent debouncing code.)

1. Construct a simple button circuit using the pin of your choice. You may use either an external or internal pull-up resistor.
2. Create a `Button` class that will manage a button interface. The constructor should take the pin that the button is connected to and store it as a data member. What other data members do you need?
3. Write a button checker function to check the button using *e*vent-driven programming.
4. Separate from the class, write a button handler function that increments a counter whenever the button is pressed and writes the total number of button presses (since the start of the program) to the Serial Monitor. Your `loop()` should look something like:

   ```
   void loop()
   {
     if(myButton.CheckButtonPress()) HandleButton();
   }
   ```

   where you'll have to write the (global) `HandleButton()` function.
5. Run your program. Does it work as expected when you press the button 10 times? (Hint: it's not actually supposed to – if it does, ask for a demo from an instructor and then carry on with the exercise.)

If the experiment is working – meaning your code is almost but not quite working right[1] – you should only need to press the button seven or eight times before it says you've pressed it ten. This is because the button is *bouncing*. At the microscopic level, every time you push the button or release it, the contacts physically bounce off one another at a very short time-scale, and the processor is fast enough to register the bounces as distinct button presses. You can *debounce* the system with hardware, software, or a combination of the two. You'll use software after you explore the phenomenon further:

1. Connect your oscilloscope to the button output and capture a button press. You may need to refresh your memory on how to use triggers. You should see something similar to that in Figure 1. Capture 5 - 10 events so that you can see what the longest bouncing duration is – the worst-case-scenario.
2. Use `delay()` to add a short delay to the code whenever there is a *change* in the button, the length of which should be more than the worst-case-scenario from your oscilloscope captures. Specifically, whenever there is a button transition (either up or down), delay 2 - 3 times as long as the longest button bounce. This will ensure that no bouncing will ever register as a button press, and still be much shorter than the time to actually press a button twice.
3. Run your program and verify that it works.

While `delay` works in this case, it is often not the best solution for many tasks. While the program is executing the `delay()`, it's unable to do anything else (interrupts excepted – more on that later). Valuable clock cycles may be wasted doing nothing. A better way is to add a variable to hold the time that the button last changed and only report a button press after a certain amount of time has passed.

[1]You may have already debounced. That's fine. Attach the oscilloscope and do the following regardless.

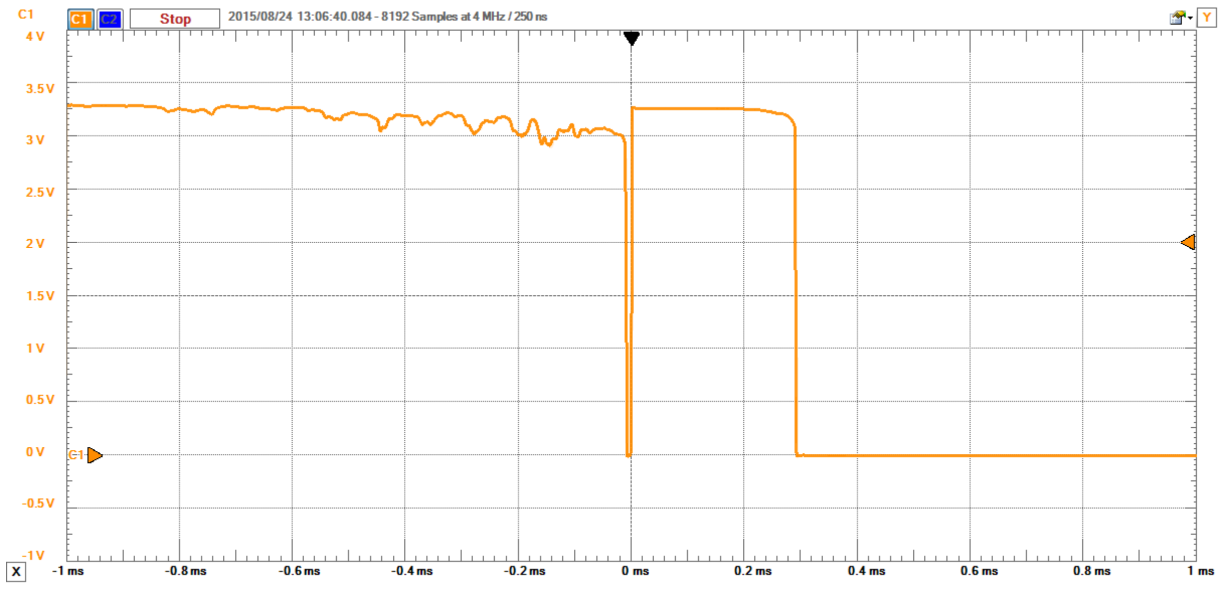

Figure 1: Pushbutton bounce phenomenon.

**Skip these two steps for now**

1. Open the `Debounce` example included with the Arduino IDE. Using the program as a template (only a template – it's not a very well written program), write a better debouncing program that doesn't use `delay()`.
2. Run your program and verify that it works.

**Show your working system to an instructor**, including an oscilloscope capture that shows the bouncing.

Instructor initials: ______________________________

## Challenge

Your first challenge project is a minimalist home alarm system. The system will need to allow activation and deactivation; sense an intruder; and sound an alarm if someone is detected. For each of these functions, you'll use the following hardware and methods:

**Activation.** To activate your system, one will merely need to press a button. Since you'll need to have time to close the door after activation, your system will wait 10 seconds after the button is pressed before it activates. You must use a software timer (not `delay()`!) for the timer, since you might want to deactivate it immediately.

**Deactivation.** A simple button will work for activation but it won't do for deactivation because it would make it easy to defeat the system – just press the button! Instead, you'll implement a numeric keypad to deactivate your system. A keypad is just a cleverly arranged set of buttons. Each of the seven pins on the keypad is connected electronically to *either* a column or a row of buttons. Pressing a key will connect two of the pins – by testing for connections you can see which button has been pressed. Typically, you would change each of the row pins `HIGH` and `LOW` in turn, and check to see if the change is registered on a column pin. **You must be careful not to cause a short circuit, however.** Specifically, you can't have two row pins as `OUTPUTS` simultaneously.

**Intrusion detection.** When the system is activated, An LED will be lit to represent a laser like you see in the movies. You'll sense the laser with a photoresistor using the internal comparator (i.e., the last circuit in the previous lab handout). When the beam is occluded, the system will alarm. Note that the laser must be turned off when the system is deactivated.

**Alarming.** Alarming will consist of a blinking LED – half-second on, half-second off – and a piezo buzzer that blares at 200 Hz. You must use a software timer for the blinking – we'll discuss the buzzer in the coming lectures. You must be able to deactivate the system using the keypad while it's alarming.

To complete the Challenge, you must do several specific tasks:

1. Develop a `Keypad` class that manages the interface with the keypad. Among other functions, it should have one called `GetDigit()`, which returns a unique value for each key when pressed.
2. Develop and implement a state machine for decoding a 3-digit (minimum) deactivation code.
3. Develop and implement a state machine for the complete system.

Be sure to follow good coding practices, as we've been discussing in class. I promise it will make your lives easier! See an instructor for hints, if you wish.

### Deliverables

For each deliverable, one per team.

**Due 2/12 at the start of class (3 points):** State diagram for system functionality. Include all states, events, actions, and guard conditions, except you may condense the keypad entry down to "receive valid keycode". Use proper nomenclature, as from the reading.

**Due 2/14 at the start of class (3 points):** State diagram for keycode entry. The reading gives some examples, but you'll have to customize them for your system.

**Due 2/19 at the start of class (10 points):** Working system. You'll have 5 minutes to get your system ready, then I'll start checking them in random order. I will go through all of the state transitions. Given the time crunch, you will not have time to fix things on the fly – you get it right or you get it wrong.

**Due 2/19 at the start of class (4 points):** Submit your code, including any custom include files (button.h, keypad.h, rc_timer.h, etc.), on collab before the start of class. I will grade for style (actual style, like using checker/handler pairs, not faux style like putting your braces in the wrong place or screwing up your capitalization). `delay()` is bad (though allowing in debouncing). Global variables should be used only where needed. Clearly define your state variables. Osv.