The Power to Control

Advanced Arduino Workshop

Updates for next time

- Instructions on using a DMM
- Appendix on HEX
- Pre-lab needs to have them come with laser checker, which needs better instruction in comparison to button
- Make it easier to turn in places where they recorded values?

Introduction

In previous tutorials, you've been introduced to Arduino, an accessible prototyping platform for electromechanical systems. Those tutorials focused on the basic functionality of Arduino – methods used to connect and read sensors or control an LED or motor. In this workshop, we'll focus on techniques for building more complex programs – programs that have to perform more than one or two tasks.

When programming a microcontroller to perform a set of tasks, it is often useful to organize the system as a *state machine*. By assigning states to the system, programming is made much more efficient (and readable!), speeding up development time. In this exercise, you'll build a basic "home alarm system," and you will be shown how to program the system as a state machine. In the process, you will also learn about *events* and using events to control your system's behavior.

Objectives

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

- Implement checker-handler construction for events,
- Implement a state machine, and
- Demonstrate their knowledge by building a home alarm system.

Preparation

The *Power to Control: Advanced Arduino* superpower sheet (linked on collab) lists several tutorials and a couple of challenges for you to do *before* you come to the workshop. Be sure to fill out the slip at the end of that document.

There is a pre-lab at the end of this document to complete. Bring your completed pre-lab to the workshop.

Events

A common function for any system is to react to a particular input with a specific response. For example, in the in-class tutorials, you pushed a button and an LED turned on or off (or dimmed). Such input-response is so common that practically every system as some form of it. But how the system registers and reacts to an input can greatly affect the system's behavior, as we'll demonstrate with an example.

Imagine you run a ball factory. You want to count how many balls roll out of each ball-making machine, so you set up a sensor that adds to a counter whenever it detects a ball passing in front of it. You might be tempted to write code that looks something like the following:

```
while the machine is running
  if a ball is present
    add one to your counter
end
```

Will this work? Why not?

The answer lies in the fact that the counter is triggered by the *presence* of a ball. If the `while` loop runs reasonably fast, then every time a ball rolls by, it will get counted *many* times and lead to a gross overestimation. A better way would be to count the *arrival* of balls:

```
while the machine is running
  if a ball is present now and was not present before
    add one to your counter
end
```

This logic will ensure that each ball is counted as it arrives at the sensor, and the counter cannot be further incremented until the ball leaves and another one arrives, guaranteeing an accurate count. In other words, the counter is triggered by the *event* of the ball arriving. An event occurs whenever there is a *change* in some condition: a button is pressed, a laser beam is broken, a timer expires.

Coding for events

To properly detect events, it is necessary to declare a variable that holds the previous value of an input and compare it to the latest value. Here's an example for our ball detector in Arduino code (which is really just C++, for those who are interested):

```
bool prevReading = false;

bool CheckBallDetector(void)
{
  bool retVal = false;
  bool currReading = ReadSensor(); //reads true if there's a ball; otherwise false
  if(prevReading == false && currReading == true)
  {
    retVal = true;
  }

  prevReading = currReading;
  return retVal;
}
```

The key is the line that looks to see if the previous reading was `false` and the current reading is `true`. Of the four possible combinations of the two variables, this is the only one that corresponds to the arrival of a ball. How would you change it if you wanted to capture the departure?

Some other rules-of-thumb to note:

- You should only read the sensor once and then use that value for the remainder of the routine. The reason is that if you read the sensor multiple times in the same routine, there is a small, but finite, chance that the value will change between readings.
- You must update the value of `prevReading` before you return from the routine.
- Though it requires a little more code, best practice is to carry a return value, in this case `retVal`, through the entire function, updating it as needed and returning it at the end.
- Checkers (and the associated handlers presented below) should all run very fast. You don't want to miss an important event because you're busy in another routine. The `delay()` function, while sometimes useful, should be used very sparingly.

When an event occurs, it needs to be handled. As such, another good practice is to structure code with *event-handler* pairs. Using our ball detector example, event-handler construction might look something like:

```
void loop()
{
    if(CheckBallDetector()) HandleDetectionEvent();
}
```

where the handler in this case is the trivial

```
void HandleDetectionEvent(void)
{
  ballCount++;
}
```

If you want more information on Arduino syntax, see the Appendix and the tutorials therein.

State machines

As noted in the introduction, it is often useful to think of an automated system as a *state machine*. A state machine consists of a set of independent states and a set of rules for how the system changes from one state to another. At any point in time the state machine can be in only one of the possible states, and that state describes the system completely. The system moves between the states in response to events, which trigger one or more *actions*. Figure 1 shows a pair of generic states, a transition between them, and the nomenclature we use for the diagrams.

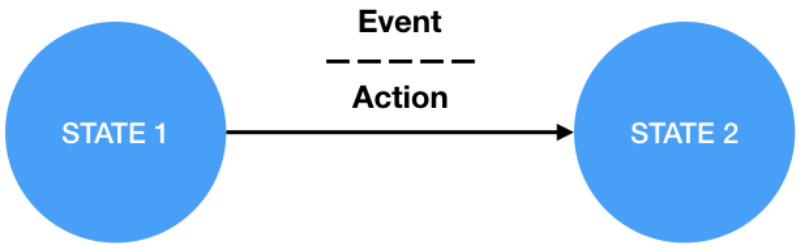

Figure 1: A pair of generic states and a transition between them.

To give a concrete example, consider an alarm clock. Figure 2 shows a state machine for a typical alarm clock with a snooze option. Starting in the `OFF` state, let's walk through the diagram. When the user sets an alarm, the system moves from `OFF` to `RUNNING`. While in the `RUNNING` state, the system checks to see of the clock time has passed the alarm time. When it does, the system starts buzzing and changes to the `ALARMING` state. While in the `ALARMING` state, if the user presses the snooze button, it does two things: turns off the buzzer and starts a snooze timer. When in the `SNOOZING` state, if the snooze timer expires, the system turns the buzzer back on and goes back to the `ALARMING` state. In any state, the user can cancel the alarm, which takes the system back to the `OFF` state.

Coding a state machine

We noted above that the system will transition from one state to another when a particular event occurs, for example the system goes from `ALARMING` to `SNOOZING` when the snooze button is pressed. Note, however, that not all events have meaning for every state. For example, in our alarm example, pressing the snooze button has no effect when the system is in the `OFF` state.

Practically, what this means is that the system only needs to respond to a given event when it's in a state where that event is meaningful. The easiest way to manage this in code is to use a *state variable* to track which state your system is in. To make the code more readable, you can use the `enum` construction, which assigns numerical values to text descriptors, as follows:

```
enum {OFF, RUNNING, ALARMING, SNOOZING};
int currentState = OFF;
```

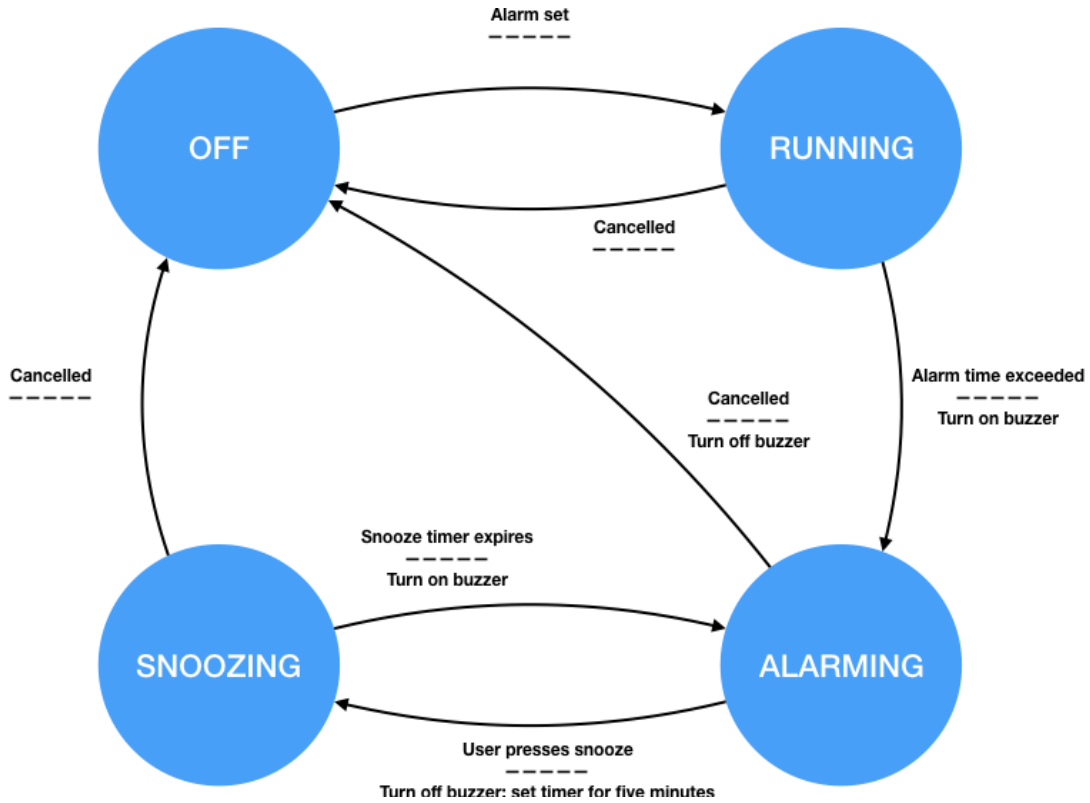

Figure 2: State machine for an alarm clock.

Here we declare a set of states with four possible values. We then declare a specific variable, `currentState`, which will be used to keep track of system's current state. From there, it's a matter of coding the state machine through conditional statements. For the event of pressing the snooze button, this might look something like the following in pseudo-code:

```
begin HandleSnoozeButton
  if current state is ALARMING
    turn off the buzzer
    start a timer for five minutes
    change current state to SNOOZING
  else if in any other state
    ignore
  end if
end
```

Note that pressing the snooze button only has meaning when in the `ALARMING` state – in other states it is ignored. For practical reasons, it is often useful to use the `switch` construct for actual code, but the syntax is a little complicated, so we won't spend time on that here (here's a tutorial, if you're interested later). Just keep in mind that it's a convenient way to organize a bunch of `if..else` statements if you ever get to that point in your project.

In the workshop

Home alarm system

Your challenge is to build and code 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 press a button. When activated, your system will turn on an LED that is pointed towards a photoresistor, which will be used for detecting an intruder – a low-tech version of the laser systems seen in the movies.

Deactivation. For deactivation, you'll need something more secure than a button. In this case you'll use an RFID reader to detect a specific tag. When deactivated the LED "laser" will turn off.

Intrusion detection. When the system is activated, if something comes between the LED and the photoresistor, occluding the "laser", the system will alarm.

Alarming. Alarming will consist a piezo buzzer that blares at 200 Hz. You must be able to deactivate the system with the RFID while it's alarming.

A skeleton code will be provided that takes care of a lot of the tedious parts – we want you to focus on implementation of the state machine.

Procedure

1. *Gently* place your RFID shield on your Arduino. Note that there are a lot of pins to get lined up, and it's easy to miss one and bend it. Don't worry too much – pins can be unbent – but the more careful you are, the better.
2. Consulting the LED/photoresisor circuits in Figure 3, do the following. Don't get ahead of yourself here, do each step in order.
   (a) Insert the photoresistor and LED into your breadboard about 3 - 4 cm apart. Gently bend the two so that the LED will shine directly on the photoresistor – they should be about 1 cm from each other when you're all done – you'll want them close enough to get a good signal while still being able to pass a pencil or a finger between them without bumping either. Once you've arranged them, try not to move either component so that the measurements you are about to take will consistent for the rest of the lab.
   (b) Connect the LED as shown, but don't connect the photoresistor to anything yet. If you connect it to other components, you might not get good readings for the next steps.
   (c) Using jumper wires (so you don't disturb the photoresistor) and a *digital multimeter* (DMM), measure and record the resistance of the photoresistor when the LED is on and when it is off. Record your values below.

   Resistance (LED on): _____

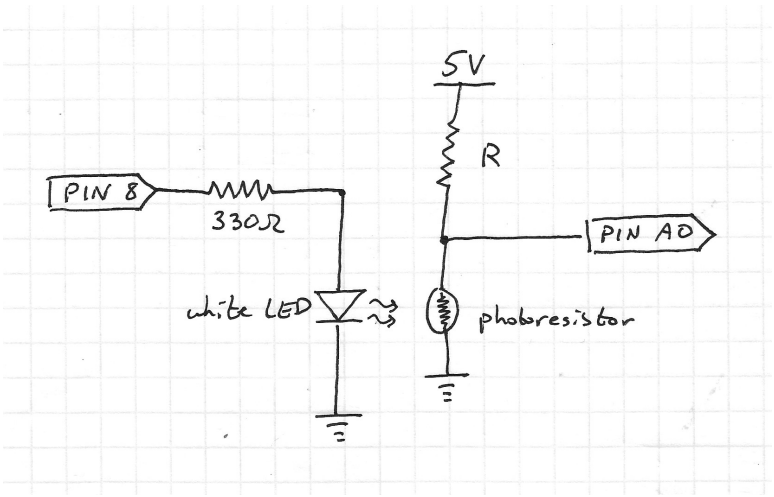

Figure 3: Circuits for the "laser".

Resistance (LED off): _____

(d) Choose a fixed resistor, R, with a value that is near the *geometric mean*[1] of the two values you measured above and complete the circuit.

(e) Use the example program `AnalogReadSerial` (or write your own) to read the ADC at the junction and print it to the Serial Monitor every 500 ms. Run some tests to determine the ADC reading with the LED on and with it off. Choose a threshold that will serve to indicate that the beam is blocked – that there is an intruder! Record your values below. How did you decide on a threshold value?

ADC reading (LED on): _____

ADC reading (LED off): _____

Threshold ADC value: _____

3. Build a button circuit on pin 7, except you don't need the pullup resistor. Instead, you'll use a nice feature of the Arduino, namely an internal pullup. To do so, you declare the pin mode as follows:

```
pinMode(buttonPin, INPUT_PULLUP);
```

This internal pullup has the same effect as the resistor you put in your breadboard, but without the hassle of extra wiring.

[1] It can be shown that doing so will give the largest voltage swing in your circuit, which makes it as sensitive as possible.

Note that you cannot use pins 2 and 3 for a button (as you did in the in-class tutorials) since they're used by the RFID shield.

4. Build the piezo "buzzer" circuit in Figure 4. Note that the piezo circuit here is different from the one you built in the pre-lab. Specifically, the current draw of the piezo puts too much strain on a pin, so we'll have you build a high current circuit similar to the one you build for the motor.

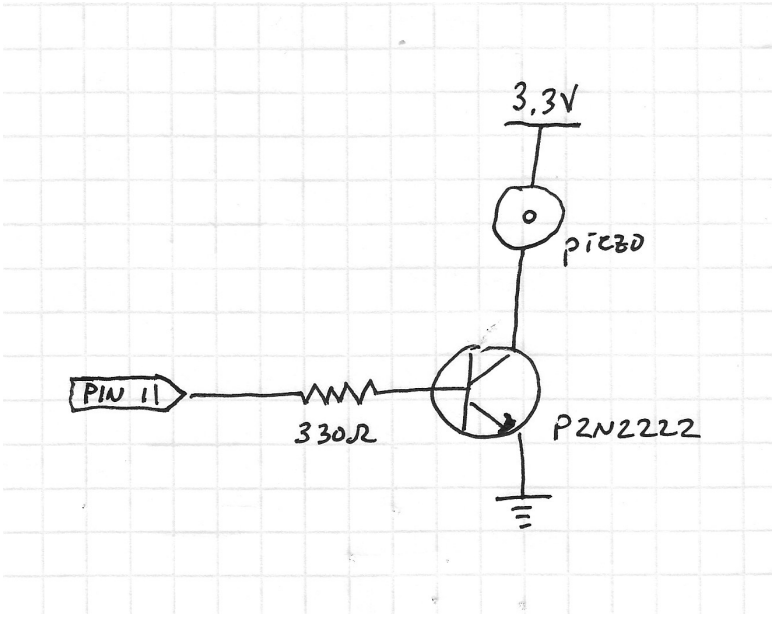

Figure 4: Circuits for the piezo "buzzer".

5. Grab the `alarm_skeleton` code from github. Because of time constraints, we're not asking you to write code from scratch.
6. Grab the RFID library from github. Install it in the library folder in your sketchbook.
7. Though it's incomplete, upload the code to your Arduino and open the Serial Monitor. If all goes well, you should be able to place a tag near the RFID antenna and get a response. Pick which tag you want to be your "key" and note the 4-segment ID.
8. Go to the top of the code and change the `targetID` to match the four values you just wrote down. Be sure to leave the `0x` part in place – this tells the Arduino that the numbers are in *hexadecimal*, which is just a short hand for writing out bytes.
9. Re-upload the code, and now it should print "`Match!`" along with the tag ID. Once you've confirmed that the tag is correct, you should comment out all of the `Serial.print` lines in `CheckForRFID()` by placing a double slash, `//`, at the beginning of each line. This will make the Serial Monitor much easier to read as we move forward.
10. Now you will fill in the missing parts of the state machine in the rest of the code. It may be helpful to put some `Serial.print()` statements to help debug. Specifically, every time there is a transition, you should print something to the Serial Monitor. See `HandleArmingButton()` in the code for an example.

(a) We've written the button checker function for you, but you need to finish the function for checking the laser. Your threshold from above will be used to check if the laser is broken. Note that you need to write it in proper event-driven form. Use the button checker as a template.

(b) We've started the button handler for you, but you'll have to fill in the actions. Note that we only explicitly address states that are specific to that event.

(c) You'll have to fill in the laser handler function. Don't forget to change the state when an event is handled!

(d) You'll also have to fill in the RFID handler. How many states does this handler apply to?

11. Test your system. If it's not working, the first thing you'll want to do is determine if the problem is in hardware or software. To check if it's hardware, use known, working codes to test the components. You can also use a digital multimeter to check voltages, for example, at a pin. If you suspect that software is the problem, try putting `Serial.print()` statements in useful places to help determine if the code is getting to where it should be.

    Of course, you can always ask an instructor for some pointers.

Once you have a working system, show it to an instructor.

Appendix: Arduino syntax

The language used in the Arduino IDE is essentially `C++`. For those who know `C++`, you can use all of its constructs, including classes, structures, overloaded operators, virtual functions, and all kinds of other things that might not mean much to you if you haven't used them before.

No worries, though! You don't need to know all of the esoteric functionality of `C++` to be able to write half-way decent programs in "Arduino". At a minimum, you need to know the different datatypes, how to write a function, `if..else` statements, `for` and `while` loops, and a few other things. Instead of re-hashing a lot of information that is out there, we'll simply point you to a decent set of beginner tutorials. Look through them on your own time for hints and pointers on whatever you need or interests you.

Pre-lab

Home alarm system

1. In the workshop, you will be building a simplified home alarm system, but first, you'll need to plan out the system using a state diagram. Your alarm system will have the following functionality:

   - To arm the system, the user will push a button,
   - When the system is armed, an LED will shine on a photoresistor,
   - If an object comes between the LED and the photoresistor while the system is armed, an alarm will sound,
   - To disarm the system, regardless of whether or not it is alarming, the user will present a specific RFID tag,
   - When disarmed, the LED will turn off, and if the system is alarming, the sound will turn off, as well.

   Draw out the state diagram for your alarm system. Be sure to include the states, events, and actions using the proper format shown in Figure 1.

2. Using pseudo-code, write out the handler function for responding to the correct RFID tag.