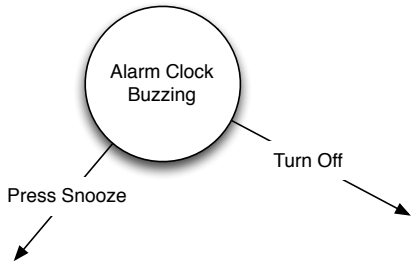
Module 0A: Security Access Device Using Ardunio

1 Objectives

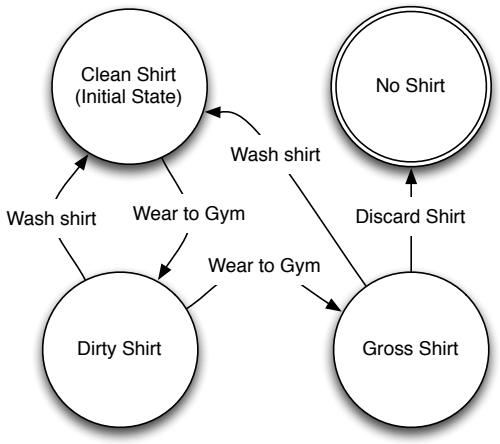
The goal of this lab is to introduce the concept of Finite State Machines (FSMs) and to demonstrate the value of modeling in application design. After modeling a simple design using FSMs, the design will be implemented using C++ on an Arduino-based embedded system

2 Introduction to Finite State Machines

There are many ways that programs can be written. One way is to use something called a Finite State Machine, or FSM. FSMs are made up of two things: some writing about what is happening and some arrows that show different choices that can be made. For example, the following is a partial FSM showing what you might do when you hear your alarm clock go off in the morning.



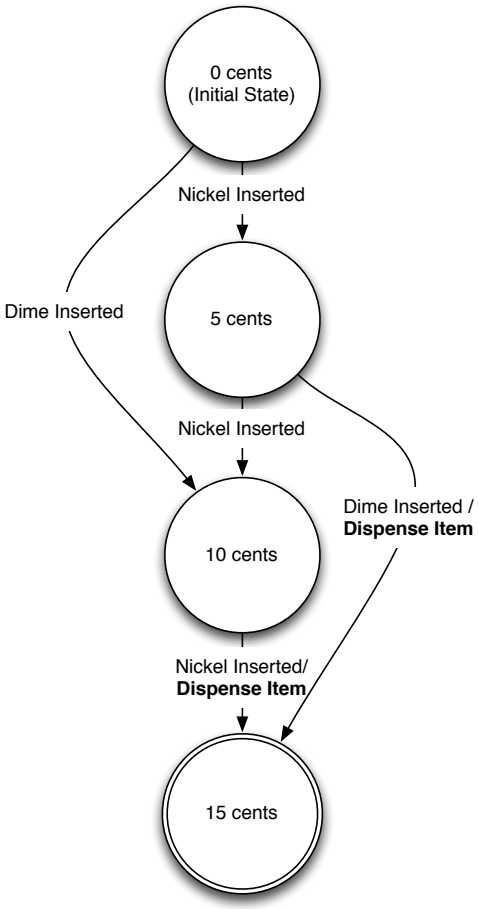
The parts of the FSMs enclosed in circles are known as *states* and the arrows are known as *transitions*. Usually, one state is labeled as the initial state; it indicates where you start from. One or more states are usually designated as final states also. They indicate where you stop. Final states are commonly drawn using double circles. The following is another, slightly more complicated, example of an FSM.



FSMs are convenient for modeling simple and complex programs. As the program executes, the finite state machine model represents the state of the program, taking transitions to other states based on certain types of inputs to the program. The program finish can be modeled as a final state in the diagram, but many applications may run continously, and their model would not have a final state.

FSM modeling is commonly a design-time activity. This means it is done prior to the majority of the implementation work. In other words, we use FSMs to capture the program states and how they change based on the inputs.

Consider the following example of an FSM that models the software for a vending machine that accepts only nickels or dimes and sells soda for 15 cents. Notice the bold items on the transitions entering the final state. Thees bold items indicate actions that should take place when the transition occurs. In this case, when transitioning into the final state, an item should be dispensed from the vending machine.



This FSM can nearly directly be converted to the following C++ implementation:

```
#include <iostream>
using namespace std;

enum MachineStates {0_CENTS, 5_CENTS, 10_CENTS, 15_CENTS};

int main() {
  MachineStates currentState = 0_CENTS;

  while(true) {
    MachineStates nextState;
    if (/* Nickel inserted into Machine */) {
      if (currentState == 0_CENTS) {
        nextState = 5_CENTS;
      }
      else if (currentState == 5_cents) {
        nextState = 10_CENTS;
```

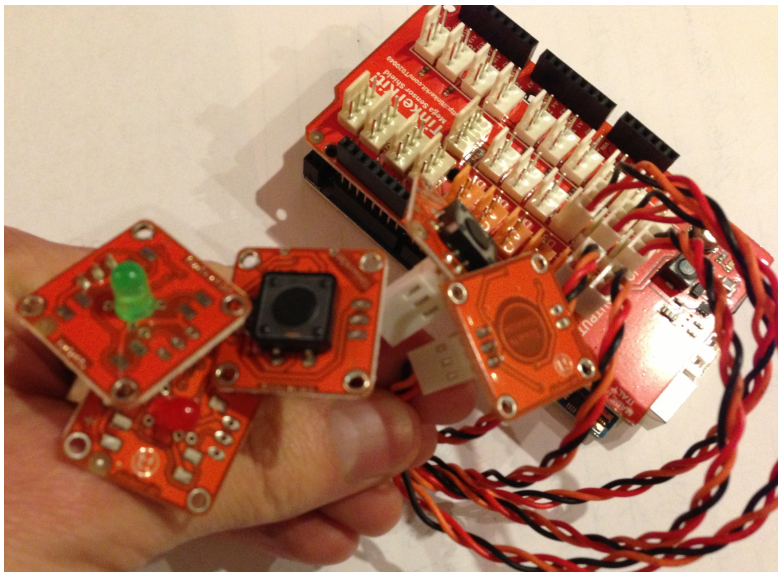
```
      }
      else if (currentState == 10_CENTS) {
        nextState = 15_CENTS;
        /* Dispense Item */
      }
    }
    else if (/* Dime Inserted into Machine */) {
      if (currentState == 0_CENTS) {
        nextState = 10_CENTS;
      }
      else if (currentState == 5_CENTS) {
        nextState = 15_CENTS;
        /* Dispense Item */
      }
    }
    currentState = nextState;
  }
}
```

Spend a few minutes studying this source code to understand how it was created from the FSM above. A C++ enumeration is used to repreesnt the set of possible states. The variable `currentState` stores the state that the FSM is currently in and the variable `nextState` stores that state that will come next. The machine states out in the initial state (`0_CENTS`, in this case, since there is no money in the machine to begin with). The next state is determined using the current state in combination with the input from the machine. If you understand this, you will be better off when you convert your FSM design to source code.

The benefit of modeling using FSMs is that the resulting software can be more robust, and most of the corner cases will already have been thought out and represented in the model. Notice that FSM for the vending machine isn't completely correct. In the above example, what would happen if the user had deposited two dimes in a row? The FSM doesn't have a transition for dealing with that scenario! If we had been more careful when designing our FSM model, we would have caught this error and resolved it before writing any code. In general, when using FSMs to model applications, it is desireable for the FSMs to have the property of *receptiveness*, where each state contains a transition for all possible inputs. This ensures that we handle all corner cases properly, and that the model has considered all possible inputs in all software states.

3 Pre-Lab Procedure

At the conclusion of this lab, you will have written the software and assembled the components for a simple security access device. The device is shown in the figure below.

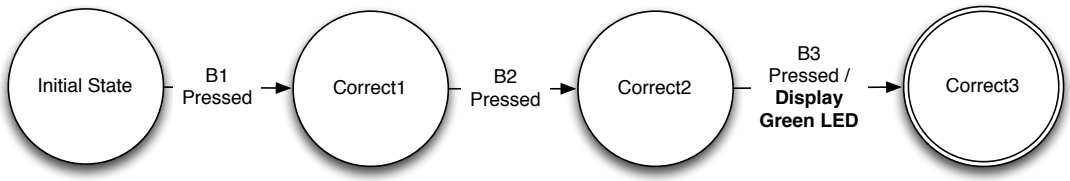


The device consists of 3 buttons, a green LED (light emitting diodes) and a red LED connected to an Arduino microprocessor. When the user presses the three buttons in the proper order, the green LED will illuminate. If the three buttons are pressed in the wrong order, the red LED will will illuminate. Carefully follow in the instructions below to model the device using an FSM, implement the model using C++ and then test the application on the provided hardware.

3.1 Model the Security Access Device

3.1.1 Step 1: Initial FSM

Let's begin by considering a simple model of this device.



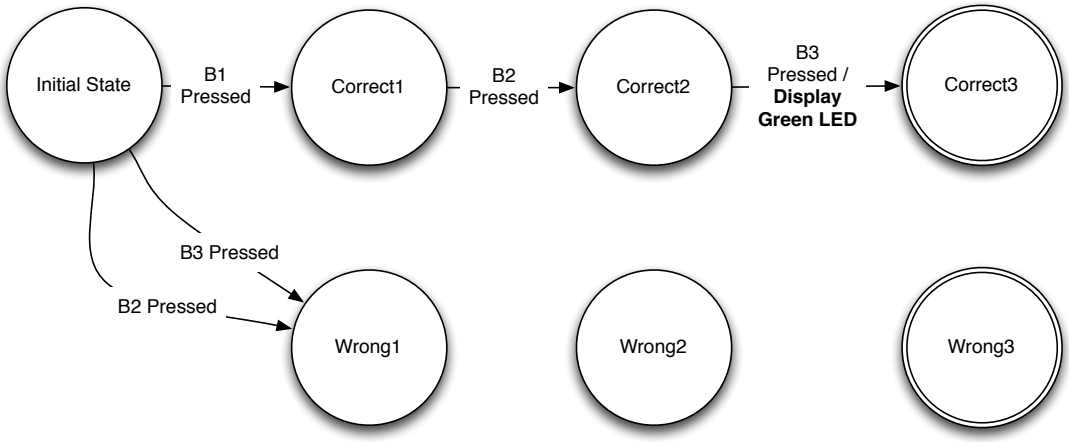
This FSM has four states, three inputs corresponding to the three buttons and one output corresponding to the green LED. In addition to the initial state, there is a state corresponding to one button having been pressed in the correct sequence (`Correct1`), one state corresponding to two buttons have been pressed in the correct sequence (`Correct2`), and a final state (`Correct3`) that corresponds to all three buttons having been pressed in the correct sequence. This FSM is good at handling a correct sequence of input (Button 1 followed by Button 2 and then Button 3), but it does not properly deal with incorrect input sequences (i.e., if Button 2 is pressed before Button 1). You can see this, because if you are in the state `Correct1`, there is no arrow corresponding to Button 2 being pressed. Let's tackle this problem next!

3.1.2 Step 2: Wrong Input at Initial State

Consider the pin number on your cell phone for a moment. When you enter an incorrect digit, does the phone immediately notify you that you entered a wrong digit? It doesn't. If it did, your phone wouldn't be very secure because it would be very easy to figure out the pin code. We want the security device to operate the same way. The user must press three buttons before a green or a red LED turns on. Let's now revisit the design of the Finite State Machine. In order to allow this behavior, we need to introduce three new

states: `Wrong1`, `Wrong2`, and `Wrong3`. These states represent the fact that the wrong sequence was pressed while keeping track of the total number of button presses. When the `Wrong3` state is reached, the red LED should be illuminated.

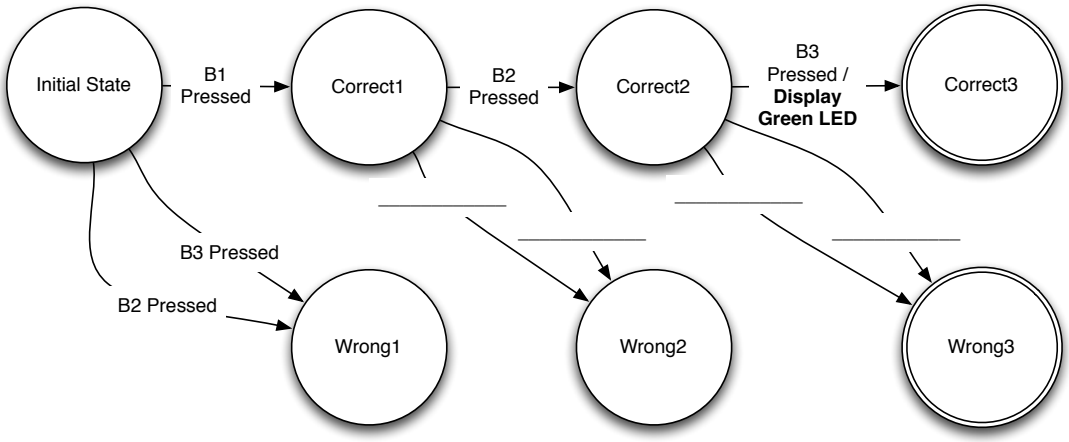
Let's focus on the Initial state and consider what should happen if the user pressed Buttons 2 or 3. In this situation, we would already know that the user did not press the buttons in the proper sequence, but we don't want to immediately display the red LED. Instead we want to wait until the user has pressed three buttons, and then turn on the red LED.



So, if we are in the initial state and the user pressed Button 2 or Button 3 first, we will transition to the state `Wrong1` which indicates that one button was pressed in the incorrect order. Now, there are transitions for each possible input from the initial state.

3.1.3 Step 3: Wrong Input After One Correct or Two Correct Inputs

Now consider the states `Correct1` and `Correct2`. In the figure below, the arrows have already been drawn that represent what happens if the user presses the wrong button after either the first button or the first two buttons were pressed in the correct order, but then the wrong button is pressed. Your task is to label these arrows with the proper inputs. Pay special attention to what should happen upon entering the `Three Wrong` state. In this case, the red LED needs to be illuminated.



3.1.4 Step 4: All Wrong Inputs

What happens if the user enters all the buttons in the wrong order. For example, if the user presses Button 2 followed by Button 1 and then Button 3, what does the state diagram indicate will happen? Write your answer in the box below.

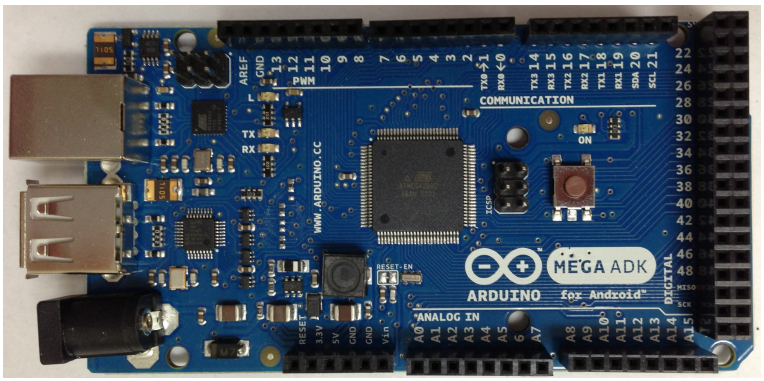
Now, in order to solve the problem that you just discovered *draw the necessary additional arrows (with the appropriate labels) to correct this behavior on the FSM figure in Step 3 above.*

When you attend lab, you must present this sheet with it correctly completed to be able to proceed to the implementation phase of the lab project.

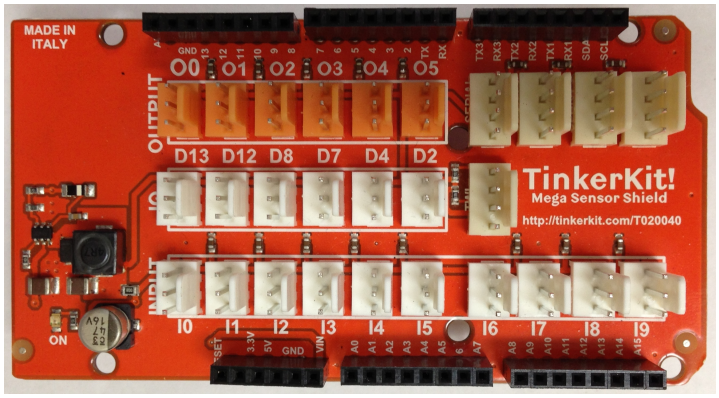
There is one more task that must be completed to prepare you for the lab. That is to familiarize you with the hardware that you will be working with.

3.2 Introduction to Arduino and TinkerKit

We have small development boards which allow you to connect devices like buttons, LEDs, accelerometers, etc. to these development boards. The development board is shown below.

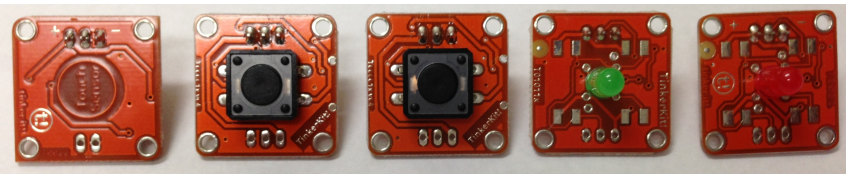


As you can see, the development board has a USB port, which you will use to load your program onto the microprocessor, and several headers to connect the TinkerKit Sensor Shield, shown below.



After you mount the TinkerKit Sensor Shield onto the Arduino Development Board, you use cable segments to connect a variety of compoments to the input and output ports that are available. The compoments that

you'll be using in the lab (touch sensor, push buttons, green LED, and red LED) are shown below. On the reverse side of the small devices are headers for connecting to the TinkerKit Sensor Shield.



That's all there is to the hardware! When you arrive at lab, you will learn how to convert your FSM design into C++ source code that you can compile and run on your Arduino development kit.

4 Lab Procedure

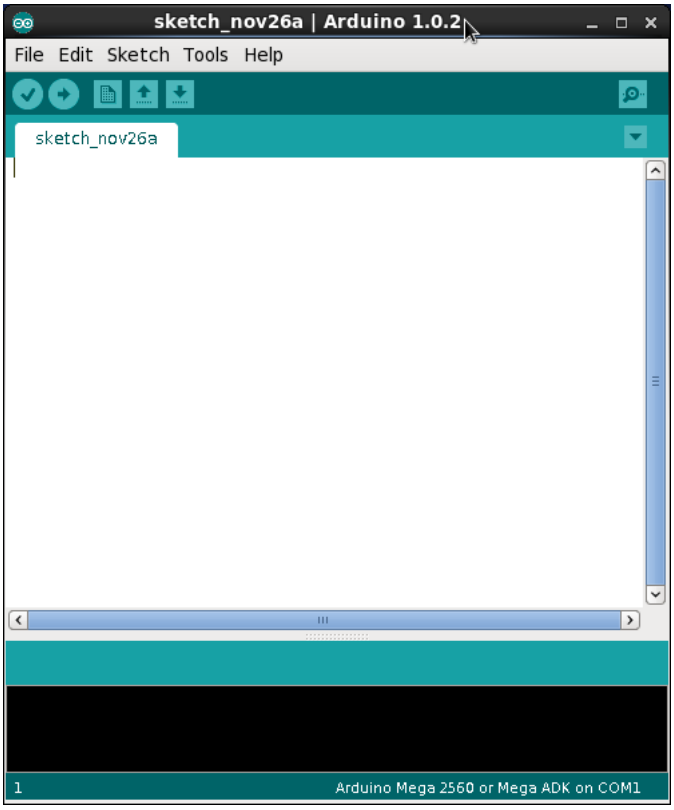
4.1 Implementing Your FSM Model

Now that you have done the hard work of creating the model of your system, and now that you know a bit about the hardware that you'll be developing for, you just have the easy part left to do: converting the model to C++ code.

4.1.1 Step 1: Start Arduino Development Environment

Your software is eventually going to run on a small microprocessor called Arduino. Therefore, you will use a special tool to write your program. This tool will eventually be used to load your program to the development board. To get started, start the Arduino development tool using the instructions provided by your lab instructor.

After you have done so properly, the following window should appear.



4.1.2 Step 2: Starting your Program

Within this window, you can write your program must like you would in Emacs. periodically, you can click on the checkmark button to cause the editor to compile your code. It is good to do this periodically to help you track down any syntax errors. Syntax errors will be displayed at the bottom of the window.

We will start by creating the general outline of a program to run on an Arduino microprocessor. In the window, type the following code:

```
#include <TinkerKit.h>

void setup() {
}

void loop() {
}
```

You can see that this is similar to other program you have written in C++. The `TinkerKit.h` file is included for special functions that we will make use of. The function `setup` is called one time when the Arduino first beings to execute your program. The function `loop` is repeatedly called by the Arduino processor several times per second. By adding code to these functions, you can cause the Arduino processor to do new and interesting things.

4.1.3 Step 3: Create your State Variable

You will next need to create a variable that will be responsible for representing what state you are currently in. We will utilize an enumerated date type for this purpose, and then create a global variable of that type. Modify your code to match the following:

```
#include <TinkerKit.h>

// The following enumeration is used to represent the possible
// states that the security device can be in.
enum SecurityState {INIT, ONE_CORRECT, TWO_CORRECT, THREE_CORRECT,
                    ONE_WRONG, TWO_WRONG, THREE_WRONG};

// The following variable represents the current state of the
// security device.
SecurityState state;


void setup() {
  state = INIT;  // Start the security device in the initial state.
}

void loop() {
}
```

4.1.4 Step 4: Create Input/Output Devices

We now have to create variables that will allow us to receive input from the buttons and to cause the LEDs to turn on and off. When we do this, we will need to specify the pins that each component will be connected to. Pay attention to this, because when you wire up the components to the SensorKit Shield, you will need to ensure that you use the proper ports. Add the additional code shown below for the input and output variables. This code should go right before the function definition for `setup`.

```
// Inputs
TKButton button1(I0);
TKButton button2(I1);
TKTouchSensor button3(I2);

// Outputs
```

```
TKLed red(O0);
TKLed green(O1);
```

4.1.5 Step 5: Create Button Input Functions

We are going to add three functions that will get executed when a button is pushed. Modify your code to make it look like the following.

```
#include <TinkerKit.h>

// The following enumeration is used to represent the possible
// states that the security device can be in.
enum SecurityState {INIT, ONE_CORRECT, TWO_CORRECT, THREE_CORRECT,
                    ONE_WRONG, TWO_WRONG, THREE_WRONG};

// The following variable represents the current state of the
// security device.
SecurityState state;

// Inputs
TKButton button1(I0);
TKButton button2(I1);
TKTouchSensor button3(I2);

// Outputs
TKLed red(O0);
TKLed green(O1);

void button1Clicked() {
}

void button2Clicked() {
}

void button3Clicked() {
}

void setup() {
  state = INIT;  // Start the security device in the initial state.
}

void loop() {
  if (button1.pressed()) {
    button1Clicked();
  }
  else if (button2.pressed()) {
    button2Clicked();
  }
  else if (button3.pressed()) {
    button3Clicked();
  }
  button1.released();
  button2.released();
  button3.released();
}
```

Inside the `loop` function we have added functions which will detect if any of the buttons have been pressed. When a button is pressed, a corresponding function will be called. These functions will be responsible for updating the state of our system. After we have detected that a button was pressed, we release the button so that it is ready to detect future presses. Remember that the `loop` function automaticallys gets called several times per second. Every time it is called, each button is tested to see if it was pressed.

4.1.6 Step 6: Implement Button 3 Transitions

Within each of the button's functions, we will implement the transitions that should occur when the corresponding button is pressed. I'll get you started with the implementation for button 3.

```
void button3Clicked() {
  if (state == INIT) {
    state = ONE_WRONG;
  }
  else if (state == ONE_CORRECT) {
    state = TWO_WRONG;
  }
  else if (state == TWO_CORRECT) {
    state = THREE_CORRECT;
    green.on();  // Turn on the green LED
  }
  else if (state == THREE_CORRECT) {
    // Nothing to do
  }
  else if (state == ONE_WRONG) {
    state = TWO_WRONG;
  }
  else if (state == TWO_WRONG) {
    state = THREE_WRONG;
    red.on(); // Turn on the red led
  }
  else if (state == THREE_WRONG) {
    // Nothing to do
  }
}
```

Study your FSM and try to figure out how this code was created from that FSM. For example, if the current state is `One Wrong` and the user presses Button 3, it is necessary to transition into the `Two Wrong` state. Similarly, if the current state is `Two Wrong` and the user presses Button 3, it is necessary to transition into the `Three Wrong` state and to turn on the red LED to indicate that the wrong button sequence was pressed.

Remember to periodically click on the checkmark button to ensure that you don't have any syntax errors.

4.1.7 Step 7: Implement the Button 1 and Button 2 Transition Functions

Using the `button3Clicked` function as a model, implement the `button1Clicked` and `button2Clicked` functions. Make sure that you implemente the transitions shown in your FSM model and don't just exactly copy the code from `button3Clicked`!

4.2 Load and Run Program on Hardware

After you have completed the above instructions and are sure that your program contains no syntax errors, you are ready to load the program onto the hardware and see if it works.

4.2.1 Step 1: Connect TinkerKit Shield to Arduino Board

If the TinkerKit shield is not already attached to the Arduino board, make the connection. When doing so, carefully align all the pins so that you do not break or bend any.

4.2.2 Step 2: Connect Input/Output Components to Tinkerkit Shiel

Obtain a labkit from the instructor and make the following connections using the provided cables:

- Push button to the `I0` (Input 0) port.
- Push button to the `I1` (Input 1) port.

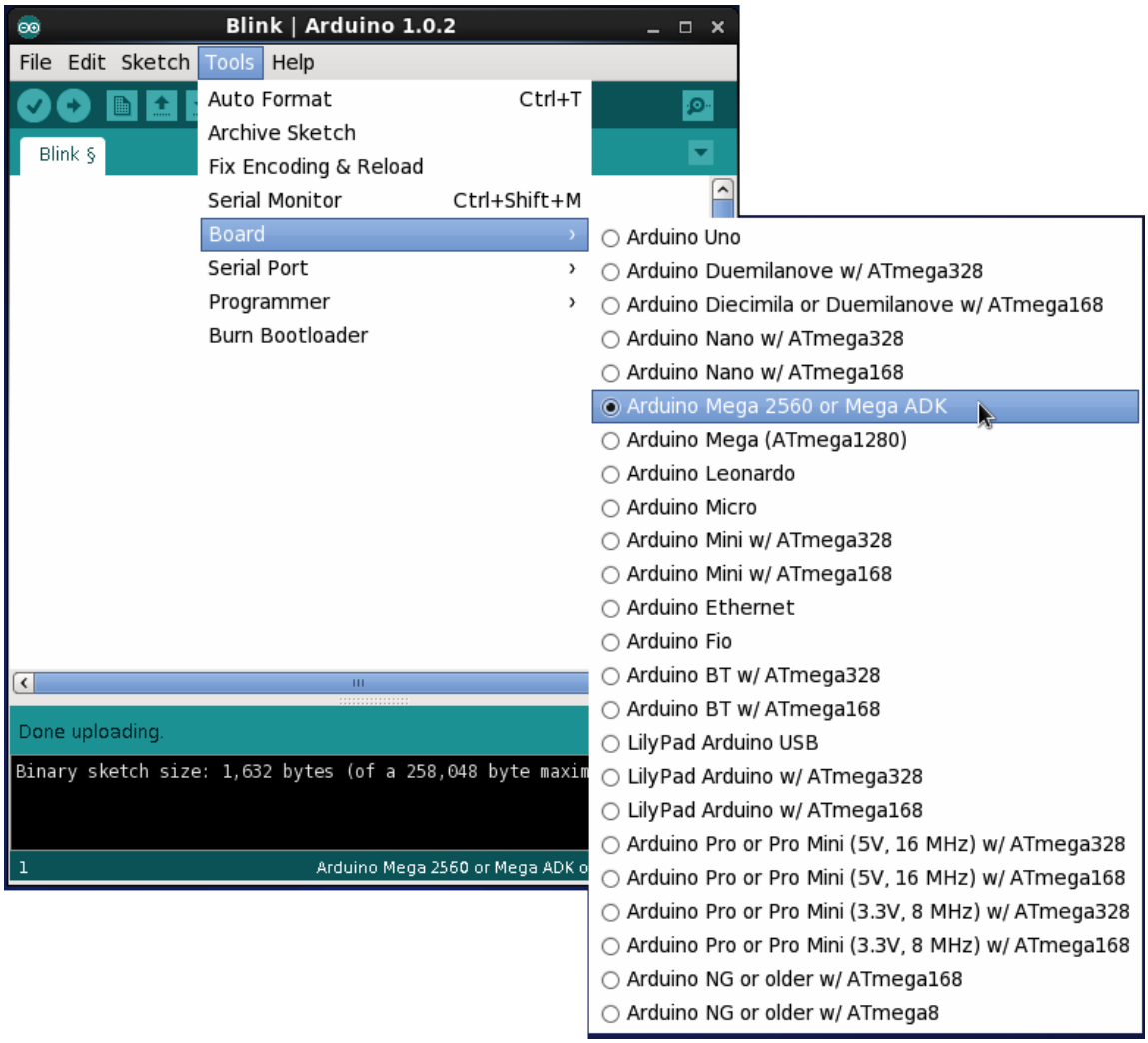
- Touch sensor to the `I2` (Input 2) port.
- Red LED to the `O0` (Output 0) port.
- Green LED to the `O1` (Output 1) port.

4.2.3 Step 3: Connect the Arduino to the Computer

Using the USB cable, connect the Arduino development board to the USB port on the front of the computer. Note that the board is receiving its power via the USB port, so it will only continue to run as long as it remains connects.

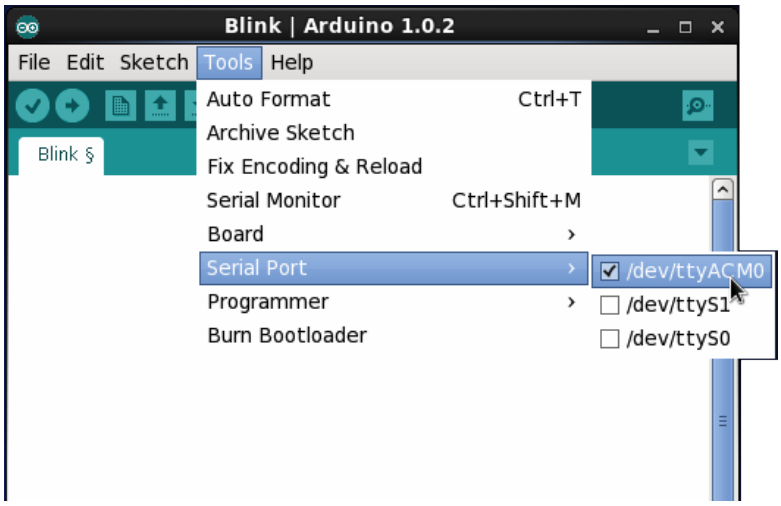
4.2.4 Step 4: Select the Appropriate Board

From the **Tools** menu of the Arduino editor, select the `Arduino Mega 2560 or Mega ADK` option under the **Board** submenu as shown in the screenshot below.



4.2.5 Step 5: Selecting the Appropriate Serial Board

From the **Tools** menu of the Arduino editor, select the /dev/ttyACM0 option under the **Serial Port** submenu as shown in the screenshot below.

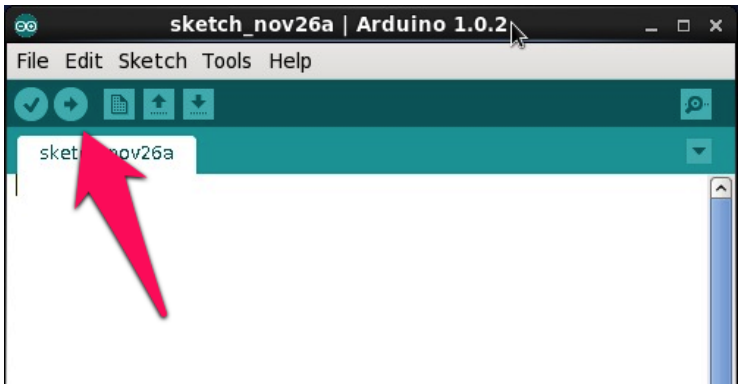


4.2.6 Step 6: Test your Program

Click on the button showing an arrow pointing to the right to compile your program, load it to the development board, and begin its execution. If all goes well, you should be able to test your application within a few seconds.

4.2.7 Sensitive Touch Sensor

Note that the touch sensor is very sensitive. Sometimes, touching it is detected as being more than one touch. As a result, if you touch it with your finger first, the red LED me appear to immediately go on. This is because three separate events were detected by the system rather than just a single touch event. For this reason, the tactile feedback provided by push buttons make them better suited for this application. Unfortunately, the kits only contain two of the push buttons though!



4.2.8 Step 7: Submit Your Work

Using the instructions provided by your lab instructor, submitted the completed Pre-Lab sheet and your source code, and demonstrate the working Security Access Device.