LAB 4: THE DIGITAL LOCK

Objective

1. Practice Arduino programming.
2. Become familiar with implementing state machines in software.

Prelab Assignment

1. Review the Arduino documentation for if/else and switch/case statements.

   http://arduino.cc/en/Reference/If

   http://arduino.cc/en/Reference/Else

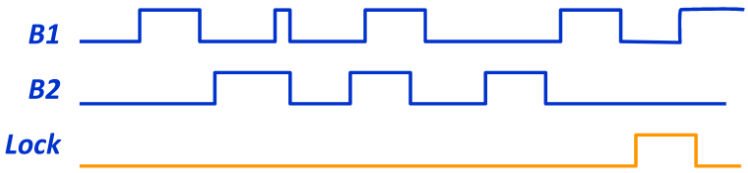
   http://arduino.cc/en/Reference/SwitchCase

2. Review Appendix A. It provides an example of implementing a finite state machine with Arduino code.
3. Draw a state transition diagram for the digital lock described below under "Desired Digital Lock Behavior" (see also Fig. 1). Your finite state machine (FSM) should include three digital (HIGH/LOW) inputs (one each for the enable switch, the pushbutton, and for matching the lock combination with the DIP switches). Your FSM should also include three digital outputs (enable LED, alarm LED, and unlock LED). See if you can combine blocks from the flowchart in Fig. 1 to implement your FSM using just four states (Hint: you don't need a START state for this exercise since the machine automatically rolls over into the ACTIVE state, independent of any input value! More than four states is not wrong, but may not be optimal.)

Finite State Machine

A finite state machine (FSM) is the logic analog of a dynamic system: its state is determined by the entire history of past input values. Outputs are determined by the current state, and transitions between states are determined by the inputs.

Consider a two-button door lock where the correct combination requires button B2 to be pushed and released, and then button B1 to be pushed and released (in that exact order), as illustrated below.



This problem has two inputs (buttons B1 and B2) and one output (lock L). The thornier problem is figuring out how many states are needed.

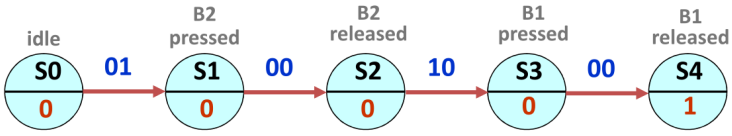
So let's reason out how many states are required to represent this sequence. First, we need an IDLE state before any buttons are pushed. Second, we need a state for when button B2 is pressed. Third, a state is required to record that button B2 has been released. Fourth and fifth, two more states are needed to record when button B1 is pressed, and then released. Once button B1 is released, the lock should unlock, so we shouldn't need any more states. If we get any unanticipated button pressed, we can go back to the IDLE state and start all over again.

We identify our needed states as S0, S1, S2, S3, and S4. In the diagram below, we show the five state identifiers in the top half of each circle, and the associated values of output L in the bottom of each circle. Note that only the final state shows the device as being unlocked.

- Inputs: *B1* and *B2* (asynchronous inputs)
- Output: *L* (*L* = 0 is lock, *L* = 1 is unlock)
- State diagram:

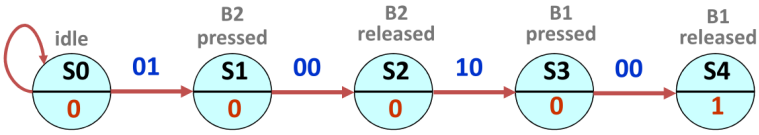
We transition from one state to the next based on input values B1 and B2. It is often easiest to start a finite state diagram by first laying out the states, then showing the desired transition path. So for this problem, we represent each possible combinations of inputs B1 and B2 as a 2-bit binary value. So if only B2 is pressed, the input is "01." If both buttons are pressed simultaneously, the input is "11." Therefore, the diagram below shows the transition from S0 to S1 when B2 is pressed. A transition from S1 to S2 occurs when button B2 is released. Similarly, we go from state S2 to S3 when button B1 is pressed, and from S3 to S4 when button B1 is released.

- Inputs: *B1* and *B2* (asynchronous inputs)
- Output: *L* (*L* = 0 is lock, *L* = 1 is unlock)
- State diagram:



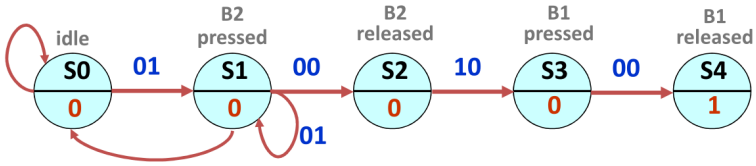
If we are in state S0, and we get any input other than button B2 being pressed, we want to stay in the IDLE state. This is represented by an arrow that loops back to the S0 state. Use of an arrow without an input label is conventionally considered to represent all input combinations not otherwise identified.

- Inputs: *B1* and *B2* (asynchronous inputs)
- Output: *L* (*L* = 0 is lock, *L* = 1 is unlock)
- State diagram:



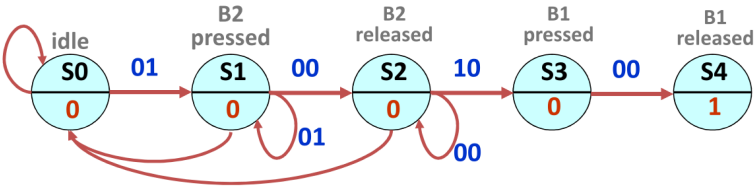
If we're in state S1, we want to stay in that state as long as button B2 remains depressed. Thus, for input "01," we stay in state S1. If button B1 is pressed (resulting in an input of either "10" or "11"), we go back to the IDLE state.

- Inputs: *B1* and *B2* (asynchronous inputs)
- Output: *L* (*L* = 0 is lock, *L* = 1 is unlock)
- State diagram:



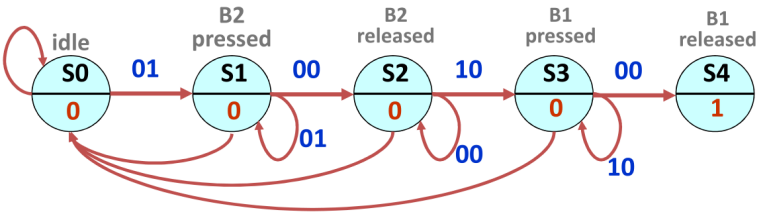
While in state S2, there is no need to change states as long as button B2 remains released. As before, however, we return to the IDLE state if button B1 is pressed.

- Inputs: *B1* and *B2* (asynchronous inputs)
- Output: *L* (*L* = 0 is lock, *L* = 1 is unlock)
- State diagram:



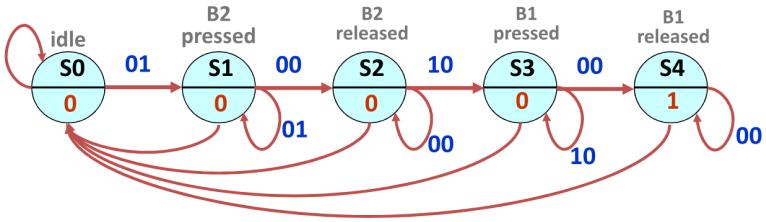
If in state S3, we stay in that state as long as button B1 remains depressed, as indicated by the loop associated with input "10." If button B2 is pressed, though, we return to the IDLE state.

- Inputs: *B1* and *B2* (asynchronous inputs)
- Output: *L* (*L* = 0 is lock, *L* = 1 is unlock)
- State diagram:



Finally, if the desired sequence of button presses has occurred, the state machine has moved to state S4. At this point the lock should unlocked, so output L should be activated (made HIGH). Also, once we are in the unlocked state, any further button presses should return us once more to the IDLE state. Thus, we are now finished drawing a finite state diagram for our two-button digital lock.

- Inputs: *B1* and *B2* (asynchronous inputs)
- Output: *L* (*L* = 0 is lock, *L* = 1 is unlock)
- State diagram:



If we desire to present our finite state machine in a more compact manner, we might prepare a state transition table, as shown below:

Next State		*Inputs: B1 B2*			
		00	01	11	10
Current State	S0	S0	S1	S0	S0
	S1	S2	S1	S0	S0
	S2	S2	S0	S0	S3
	S3	S4	S0	S0	S3
	S4	S0	S0	S0	S0

Desired Digital Lock Behavior

You will construct a FSM in this exercise to simulate a digital lock having the following desired behavior:

- When the enable switch is OFF, the lock can be set with the four DIP switches.
- When the enable switch is ON, the lock is locked and can only be unlocked with the correct combination of DIP switches.
- Pressing the pushbutton triggers the checking of the combination.
- The wrong combination triggers the alarm.
- The correct combination unlocks the door.

This behavior is shown as a flowchart in Fig. 1. The blocks shown may or may not correspond to blocks in your finite state diagram.

Procedure

1. **Creating a Digital Lock**
   (a) Build a circuit to implement the digital lock. You will need one pushbutton, one enable switch, four DIP switches, three LEDs for indicating status and four LEDs for indicating the lock combination. The 4-bit binary code is to

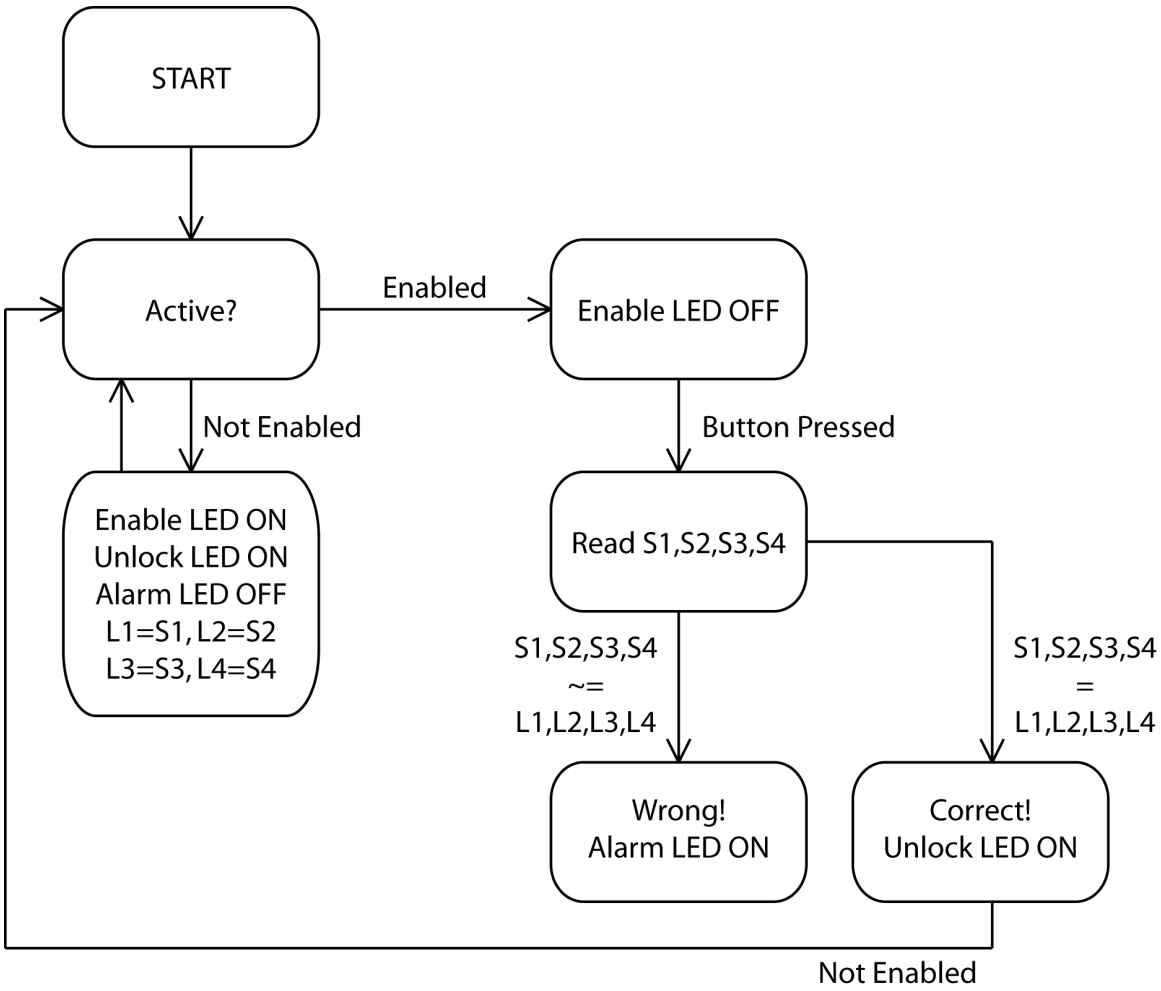


Figure 1: Digital lock logic as a flowchart.

be shown with four red LEDs. The enable status is shown with an amber LED, the lock/unlock status is shown with a green LED, and the alarm status is shown with a red LED.

(b) Using your FSM from the Prelab, write code to implement the digital lock as a state machine. Sample code is included in Appendix A.

(c) Verify the operation of the digital lock. Does it work as expected? Include the final state transition diagram and state transition table in your report.

2. **Creating a Digital Lock Using Stateflow**

Repeat the exercise using Stateflow (see Appendix B) to program the state machine.

Verify the operation of the digital lock. Does it work as expected? Is this method easier or harder than writing the code for the Prelab Assignment? What are some advantages and disadvantages of each method? Include screenshots of your Stateflow chart and Simulink diagram in your report.

Appendix A Sample State Machine Code

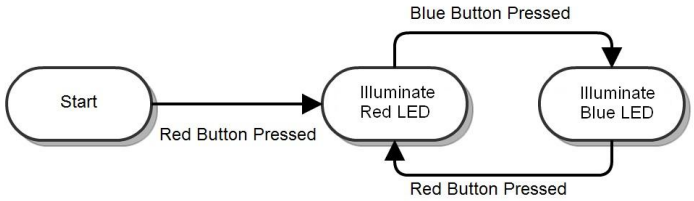


Figure 2: Sample state machine diagram.

A finite state machine (FSM) changes its behavior (output) depending on its current state. Therefore, to implement a FSM in Arduino code, we need to track the machine state, activate the appropriate outputs for that state, and then read the inputs to see if a state transition is required. In the description that follows, code snippets will be shown as various parts of the FSM are described. The complete program, in its entirety, is presented at the end of this Appendix.

For the example shown above, we define three states for our state machine: `START`, `RED_LED_ON`, and `BLUE_LED_ON`. There is nothing special about these names; however, it is good practice to select names that are easy to understand. Also, these names do not have to be UPPERCASE, but it is a common practice to capitalize variables that have fixed values. In the current example, the names `START`, `RED_LED_ON` and `BLUE_LED_ON` can be defined as unique integer values that identify the active machine state. Thus, our Arduino code associates a distinct integer with each of the state names:

```
//define state names and associated integer values
const START 100
const RED_LED_ON 101
const BLUE_LED_ON 102
```

You'll note that the assigned integer values start with 100, rather than 0. There's nothing special about the integer we start with; 0, 1 and 2 would have worked also. However, just to make sure that we don't confuse state numbers with pin numbers, we elect here to associate our three states with integers starting from 100 (far above the number of pins on an Arduino board).

We can also infer from the flowchart that two LEDs need to be illuminated, so we arbitrarily select descriptive output names, choosing `RED_OUT` and `BLUE_OUT`. Our code associates these names with specific pin numbers (8 and 9) that will later be assigned as Arduino board outputs.

```
//assign output pin numbers
const RED_OUT 8
const BLUE_OUT 9
```

Further, we can deduce that when we are in state S0, neither of the outputs should be on. When we are in state S1, we should active output pin RED_OUT, which in turn should supply voltage to a red LED. And when we are in state S2, we should activate output pin BLUE_OUT, thus illuminating a blue LED. So our code needs to activate the output pins based on the machine state. It will look something like this:

```
//generate appropriate outputs
switch(state){
case START:
    digitalWrite(RED_OUT, LOW);   // turn red LED off
    digitalWrite(BLUE_OUT, LOW);  // turn blue LED off

    ...
}
```

Transitions between FSM states depend on the current inputs. Based on the flowchart above, we can see that there are two inputs; a red button and a blue button. Therefore, we define input names and associate them with specific pin numbers:

```
//assign input pins
const RED_BTN 2
const BLUE_BTN 3
```

We will also find it helpful to track input conditions, so we define values that can serve as flags to indicate when a button has been pressed:

```
//assign input conditions
const RED_PRESS 200
const BLUE_PRESS 201
const NO_PRESS 202
```

As the program is started, we need to define the initial state:

```
//create state variable and initialize it to START state
int state = START;
```

Next, we assign input and output pins on the Arduino board. This is done in a routine called `setup()`. (The `void` keyword simply indicates that this function does not return any value to the calling routine.)

```
void setup(){
  pinMode(RED_BTN, INPUT);
  pinMode(BLUE_BTN, INPUT);
  pinMode(RED_OUT, OUTPUT);
  pinMode(BLUE_OUT, OUTPUT);
}
```

Now that the setup is done, we go into a endless loop, scanning the inputs for values that dictate a machine state change:

```
void loop(){
//read inputs
if(digitalRead(RED_BTN)){
    input = RED_PRESS;
    delay(100); //for debouncing
}
else if(digitalRead(BLUE_BTN)){
    input = BLUE_PRESS;
    delay(100); //for debouncing
}
else {
    input = NO_PRESS;
}
```

Note that we are ignoring the condition where both the red and blue buttons are pressed at the same time. Do you think this is a problem?

Based on the current state, the appropriate outputs are set, and then the necessary state transitions are implemented with code that looks something like this:

```
switch(state){
    case START:
        digitalWrite(RED_OUT, LOW);  // turn red LED off
        digitalWrite(BLUE_OUT, LOW); // turn blue LED off

        //now evaluate input condition to determine next state...
        switch(input){
            case RED_PRESS:
                state = RED_LED_ON; //go to RED_LED_ON state
                break;
            case BLUE_PRESS:
                state = START; //stay in START state
                break;
            case NO_PRESS:
                state = START; //stay in START state
                break;
        }
        break; //jump out of current switch statement
```

```
    ...
}
```

Note the use of `break` statements to keep the `switch()` function from evaluating code beyond a single case; these are important! Also, in the previous snippet, since the `BLUE_PRESS` and `NO_PRESS` input conditions don't cause a state change, we can drop those lines from our program, resulting in shorter, but equally effective code:

```
switch(state){
    case START:
        digitalWrite(RED_OUT, LOW);  // turn red LED off
        digitalWrite(BLUE_OUT, LOW); // turn blue LED off

        //now evaluate input condition to determine next state...
        switch(input){
            case RED_PRESS:
                state = RED_LED_ON; //go to RED_LED_ON state
                break;
        }
        break; //jump out of current switch statement

    ...
}
```

In fact, since there's only a single condition to be evaluated, we could shorten the code even further, using an `if` statement in place of the `switch` function:

```
switch(state){
    case START:
        digitalWrite(RED_OUT, LOW);  // turn red LED off
        digitalWrite(BLUE_OUT, LOW); // turn blue LED off

        //now evaluate input condition to determine next state...
        if (input==RED_PRESS){
            state = RED_LED_ON; //go to RED_LED_ON state
        }

    ...
}
```

Now that the outputs have been updated, and the desired state has been assigned, we jump back up to the top of the `loop()` routine and start the process all over again.

So here is the Arduino code in its entirety:

```
//define state names and associated integer values
const START 100
const RED_LED_ON 101
const BLUE_LED_ON 102

//assign names to possible input conditions and give them unique values
const RED_PRESS 200
const BLUE_PRESS 201
const NO_PRESS 202

//assign input pins
const RED_BTN 2
const BLUE_BTN 3

//assign output pin numbers
const RED_OUT 8
const BLUE_OUT 9

//create state variable and initialize it to START state
int state = START;

//create input variable and initialize
int input = NO_PRESS;
```

```
void setup(){
  pinMode(RED_BTN, INPUT);
  pinMode(BLUE_BTN, INPUT);
  pinMode(RED_OUT, OUTPUT);
  pinMode(BLUE_OUT, OUTPUT);
  digitalWrite(RED_OUT, LOW);   //turn red led off to start
  digitalWrite(BLUE_OUT, LOW);  //turn blue led off to start
}

//begin continuous loop
void loop(){
  //read inputs
  if(digitalRead(RED_BTN)){
    input = RED_PRESS;
    delay(100); //for debouncing
  }
  else if(digitalRead(BLUE_BTN)){
    input = BLUE_PRESS;
    delay(100); //for debouncing
  }
  else {
    input = NO_PRESS;
  }

  //generate appropriate outputs and evaluate state transitions
  switch(state){
    case START:
      digitalWrite(RED_OUT, LOW);   // turn red LED off
      digitalWrite(BLUE_OUT, LOW);  // turn blue LED off
      if (input==RED_PRESS) {
        state = RED_LED_ON;
      }
      break;

    case RED_LED_ON:
      digitalWrite(RED_OUT, HIGH);   // turn red LED on
      digitalWrite(BLUE_OUT, LOW);   // turn blue LED off
      if (input==BLUE_PRESS) {
        state = BLUE_LED_ON;
      }
      break;

    case BLUE_LED_ON:
      digitalWrite(RED_OUT, LOW);    // turn red LED off
      digitalWrite(BLUE_OUT, HIGH);  // turn blue LED on
      if (input==RED_PRESS) {
        state = RED_LED_ON;
      }
      break;
  }
}
```

Appendix B Simulink Stateflow

1. Load Simulink and the Simulink Library Browser.
2. Navigate to Stateflow in the Simulink Library Browser.
3. Add a Chart block to your Simulink diagram.

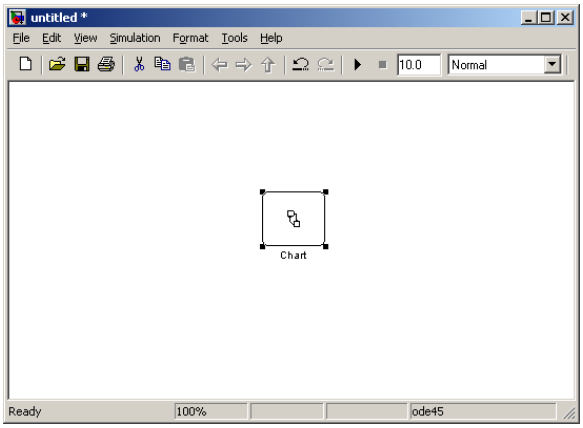


Figure 3: Simulink diagram with chart block.

4. Double-click the chart block.
5. Insert blocks for each state. Insert a "default transition" to the starting state. Click on a state and drag to another state to create transitions between states.

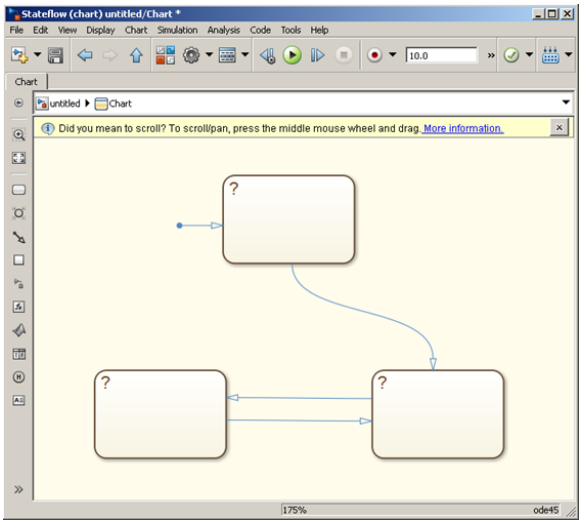


Figure 4: Stateflow with transitions and states added.

6. Click on the "?" in a state. Label the state and create a label and actions for the state. The "en:" action triggers when the state is entered. The "du:" action triggers while the state is active, but not the first time that the state is entered. Double click on a transition. Click the "?" and write in a transition condition.

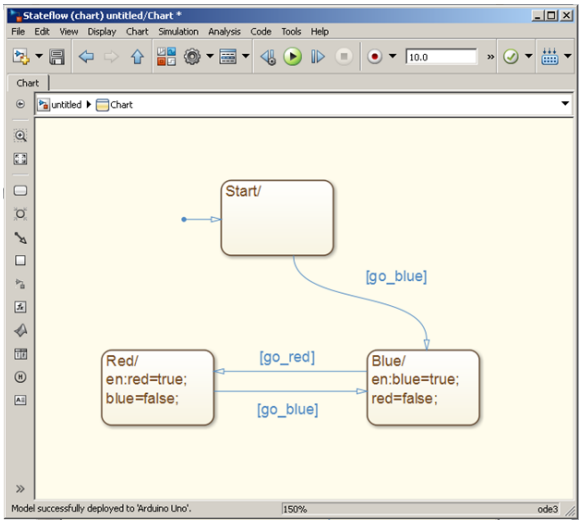


Figure 5: Stateflow with transitions and states labeled.

7. Navigate to View → Model Explorer. Use the "Add Data" button to create inputs, outputs and local variables. These can be global to stateflow diagram or local to a particular state. If you want the variable to be an input, or output compatible with the Arduino Input and Output Simulink blocks, use the Boolean variable type for digital inputs, or change it to 'Inherit: Same as Simulink'. Note: by clicking on Chart → Parse Chart, Matlab will automatically create variables and ask you to assign them as inputs and outputs along with their type.

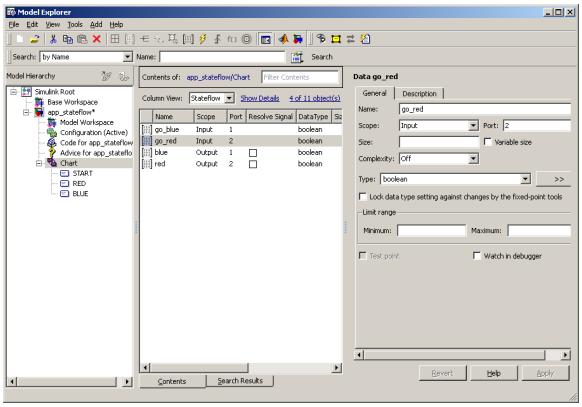


Figure 6: Stateflow model explorer with inputs and outputs added.

8. Build the rest of the Simulink diagram.

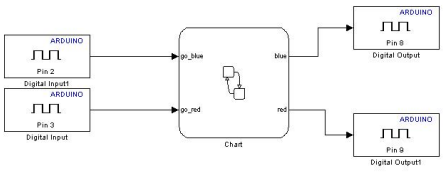


Figure 7: Stateflow chart with digital inputs and outputs.

9. Build the Simulink diagram to the Arduino board.