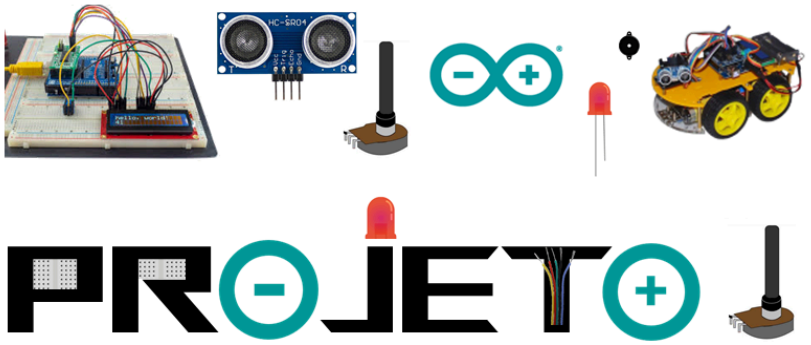

Licenciatura em Engenharia Eletrotécnica e de Computadores

Laboratorial assignment # 3

LDR, Push Buttons and State Machines

Components list:

- Arduino UNO
- 1 USB cable
- 1 white breadboard
- 1 LDR sensor **+ 10K Ohm resistor**
- 5 LEDs: 2 red, 2 green, 1 yellow
- 7 220 Ohm resistors
- 1 Push button
- Wires
- Software: Arduino IDE

1. Light Dependent Resistor (LDR)

A new sensor used in this work is the Light Dependent Resistor (LDR). It changes its resistance depending on the intensity of light detected. To mount this sensor, use the circuit shown in **Figure 1**.

The voltage read by the analog pin depends on the luminosity at the LDR. If the luminosity is low (dark ambient) the LDR resistance is high and then the voltage at its terminals is also high (near +5V). If the luminosity is high, the LDR resistance is low and then the voltage at its terminals is low (near +1V). Notice that the value read at the Arduino analog input pin is converted within the range 0 (+0V) to 1023 (+5V).

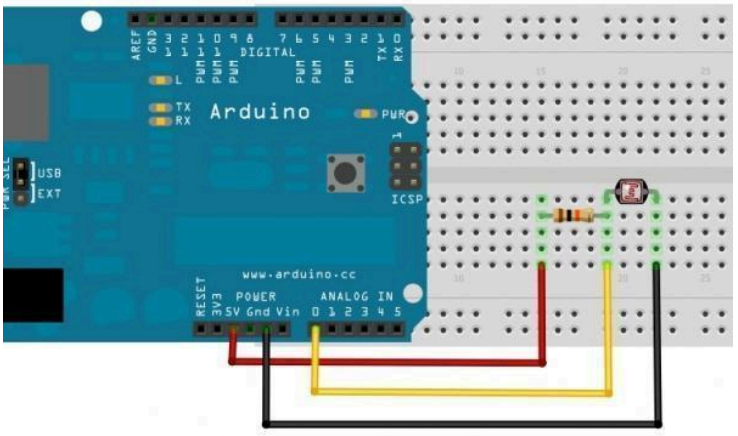

Figure 1. Circuit with a Light Dependent Resistor (LDR) and a 10K Ohm resistor.

Serial communications

The serial port of the Arduino can communicate with the computer to send and receive data.
This will be useful to check the value read by the LDR.
The operation of this serial communication is made using the following functions:

- **`Serial.begin(baud_rate)`**—indicates to the Arduino the data rate speed used to communicate. It should be set in the initial **`setup()`** function;
- **`Serial.print(value)`**—send the **`value`** through the serial port. It is able to send variables in several formats, including characters or strings;
- **`Serial.println(value)`**—send the **`value`** through the serial port and change line (introduces a '\n' at the end).

Mount the LDR sensor and test its luminosity range by sending the analog value read to the serial port. Cover the LDR with your finger or with a small paper cap and observe the differences. Depending on the time of day, your professor can turn the lab lights off and on, so that you can observe the behavior of the sensor. What's the point (and need) of using a serial resistor together with the LDR?

2. Push Buttons

Push buttons allow us to interact with our circuit and our code. When the push button is **unpressed,** there is no connection between the two terminals of the push button (the circuit is **open**). When the button is **pressed**, the circuit is **closed** - its two legs become electrically connected.

If we install the push button as represented in Figure 2, we can read the button state using a digital input. When the push-button is unpressed (open), the pull-down resistor causes the digital input pin of the Arduino to be connected to GND - the Arduino reads a digital **LOW.** When the button is pressed (closed), the digital input is connected to the 5V supply - the Arduino reads a digital **HIGH**.

Note that you can also wire the circuit the opposite way (with a pullup resistor keeping the input HIGH and going LOW when the button is pressed).

For a simple test of the Push button, mount the circuit below and upload and test the example code given in the Arduino IDE by navigating to:

File > Examples > Digital > Button

File > Examples > Digital > Debounce

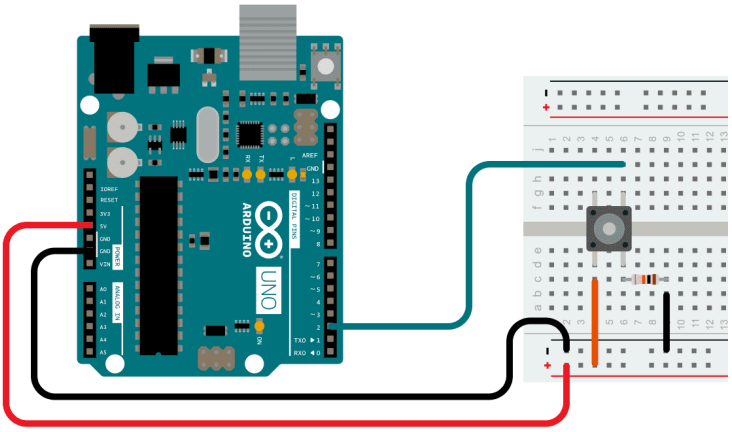

Figure 2. Circuit with a Push Button and a 10K Ohm resistor.

Can you explain the output? What is the purpose of the debounce functionality?

3. State machines

Very often, systems have different behaviors depending on their current state, and specific conditions cause the system to change its state.

Take this example: a garden light might be programmed to turn on automatically when motion is detected (and turn off after some time) during the night, but never turn on during the day. As you can see, the system has two very distinct behaviors, depending on its current state (night or day).

When using microcontrollers, the most common way to implement different operating states is by using state machines. In this lab, you will learn to use state machines to create dynamic systems with ease.

State machines can be represented in many ways, but a common representation is shown below. Each component of the state machine representation is described in green.

As long as it is well defined, a state machine can easily be implemented in code, making it simple to separate the various system behaviors to be executed in each state.

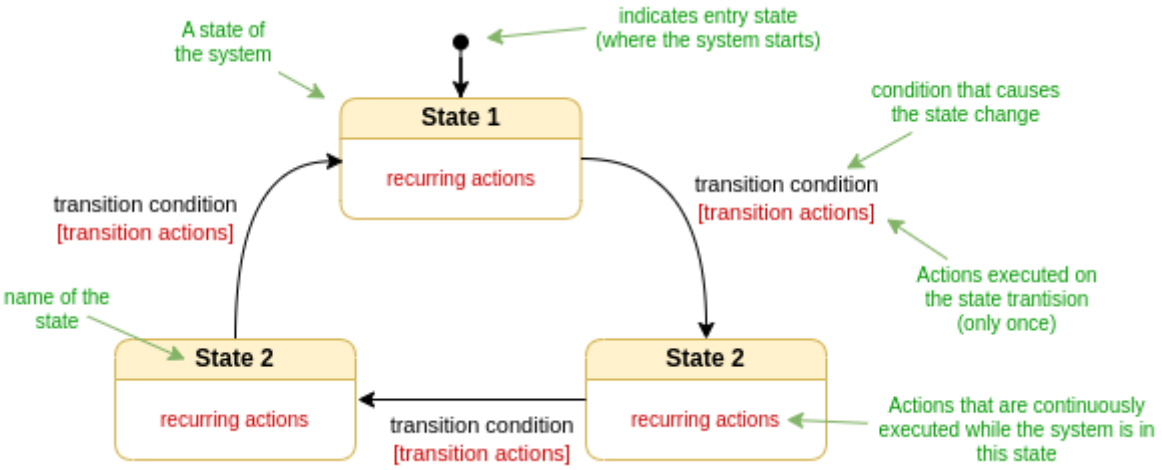

Figure 3. Basic state machine representation.

3.1 Basic state machine for LED switching

Consider this simple example: a system has 3 LEDs (red, yellow and green) and a push button. When the button is pressed, the program changes the active LED, in the following order RED→YELLOW→GREEN and then goes back to RED.

This system can easily be modeled with the following state machine:

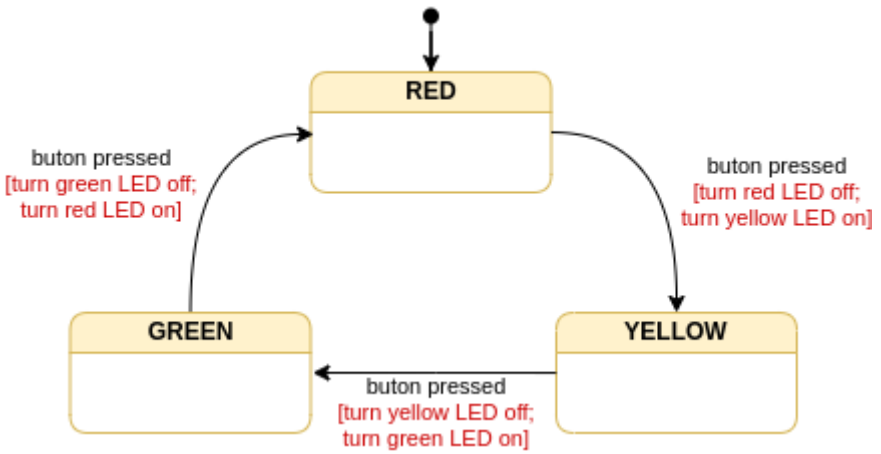

Figure 4. State machine for LED switching.

As you can see, this system does not use any recurrent actions inside the states, because the LED states only need to be modified when the state is changed.

The most common way to implement a state machine in microcontrollers is by using a switch-case construct, and a set of predefined states (typically defined using an enum).

The code below shows a possible implementation of the state machine shown in Figure 4. Analyze the code to understand how the state machine is implemented. Mount the circuit in Figure 5 and test the execution of the program. Discuss it with your professor.

How do you think the program would behave if debounce was not implemented on the button?

```
// Pin definitions
#define BUTTON_PIN  (10)
#define LED_RED     (7)
#define LED_YELLOW  (6)
#define LED_GREEN   (5)

// Definition of main program states
// Each defined keyword corresponds to an increasing
number (0,1,2)
enum STATE_ENUM
{
 RED,
 YELLOW,
 GREEN
} system_state;


// Variables used for button and debounce
int buttonState;             // the current reading
from the input pin
int lastButtonState = LOW;  // the previous reading
from the input pin
unsigned long lastDebounceTime = 0;  // the last time
the output pin was toggled
const unsigned long debounceDelay = 50;    // the
debounce time; increase if the output flickers


// Function to detect the press of a button, with
debounce (adapted from the Arduino Debounce Example)
// -> returns true if the button was pressed (after
the debounce time)
bool buttonPressed() {
 int reading = digitalRead(BUTTON_PIN);
 bool pressed = false;

 // If the switch changed, due to noise or pressing:
 if (reading != lastButtonState) {
   // reset the debouncing timer
   lastDebounceTime = millis();
 }

 if ((millis() - lastDebounceTime) > debounceDelay) {
   // if the button state has changed:
   if (reading != buttonState) {
     buttonState = reading;

     // only toggle the LED if the new button state is
HIGH
     if (buttonState == HIGH) {
       pressed = true;
     }
   }
 }

 // save the reading. Next time through the loop,
it'll be the lastButtonState:
 lastButtonState = reading;
  return pressed;
}
```

```
void setup() {
 //setup your pins
 pinMode(BUTTON_PIN, INPUT);
 pinMode(LED_RED, OUTPUT);
 pinMode(LED_YELLOW, OUTPUT);
 pinMode(LED_GREEN, OUTPUT);
  // define initial state
 system_state = RED;
 digitalWrite(LED_RED, HIGH);
 digitalWrite(LED_YELLOW, LOW);
 digitalWrite(LED_GREEN, LOW);
  //initialize button state
 lastButtonState = digitalRead(BUTTON_PIN);
}

void loop() {
 // STATE MACHINE
 switch (system_state){
   case RED:
     // RECURRING CODE: executed at every loop execution
     // <CODE>

     // CONDITIONS FOR STATE TRANSITION - different state
transitions should be included as different ifs
     if(buttonPressed()){ //change state when button is
pressed
       // CODE TO EXECUTE ON THE STATE TRANSITION:
executed only on the transition
       digitalWrite(LED_RED, LOW);
       digitalWrite(LED_YELLOW, HIGH);

       // CHANGE TO THE NEW STATE
       system_state = YELLOW;
     }
     break;

   case YELLOW:
     if(buttonPressed()){
       digitalWrite(LED_YELLOW, LOW);
       digitalWrite(LED_GREEN, HIGH);

       system_state = GREEN;
     }
     break;

   case GREEN:
     if(buttonPressed()){
       digitalWrite(LED_GREEN, LOW);
       digitalWrite(LED_RED, HIGH);

       system_state = RED;
     }
     break;

   default:   // should never get here -> safety
     system_state = RED;
     break;
 }
}
```

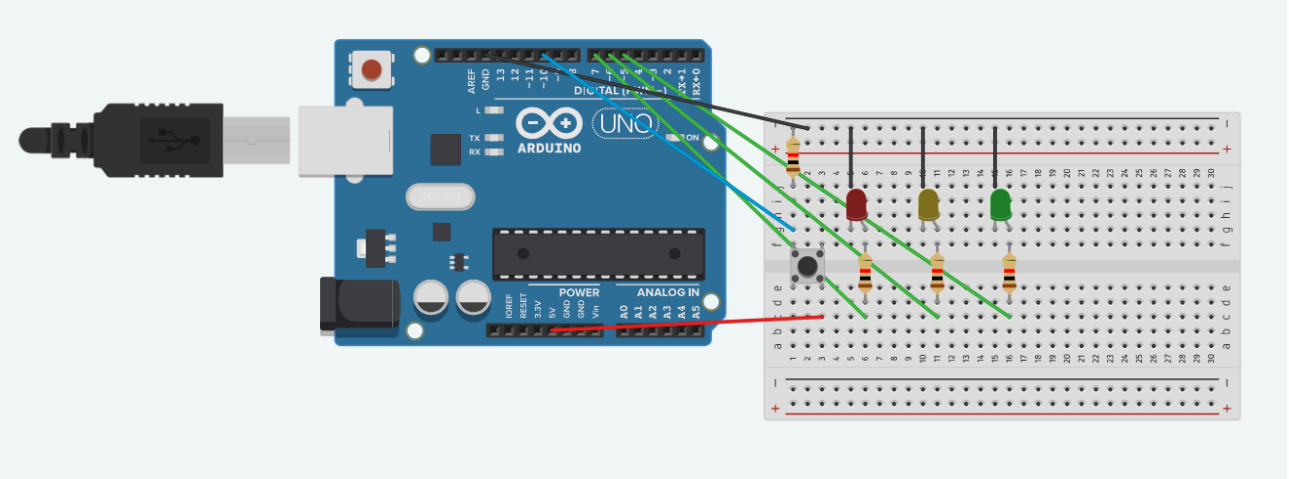

Figure 5. Circuit for testing the LED switching state machine.

3.2 Adding new states

Modify the previous code to include two additional states: ON and OFF. In these two states, all LEDs should be on or off, respectively. The ON state should be the entry state of the system. The system should cycle through states in the following order: ON→OFF→RED→YELLOW→GREEN and back to ON.

3.3 Blocking code on state machines

A major advantage of defining state machines is that you can create several “tasks” to be executed “in parallel”. For example, you can define 2 independent state machines and have them run “in parallel” (both are executed in the loop task - the program checks the state of each one and takes the corresponding actions in turns).

Even though you can include any code inside a state machine, it is undesirable to use blocking code inside the states. Blocking code is any code that blocks the execution of the program for “long” periods of time. If the code inside a state machine blocks the execution, it not only blocks that state machine, but also any other code outside it, which compromises the operation of the program.

For example, consider a simple blinking LED example, in which the LED stays on for 3 seconds and off for 3 seconds, as shown in Figure 5.

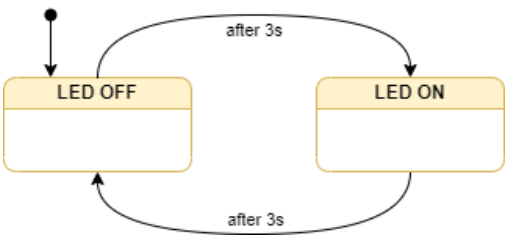

Figure 5. Simple blinking LED state machine.

If you implement the 3 second wait using a delay, any other code will be prevented from running during the delay. Hence, the waiting time should be implemented in a non-blocking way (typically using millis()).

The example below does the following:

- The state machine in Figure 5 (blinking LED), is implemented in the function "`Blink_state_machine()`".
- The program continuously reads the value of an LDR and prints it to the serial console.

```
#define LED_pin    (4)

// Definition of main program states
// Each defined keyword corresponds to an increasing number (0,1,2)
enum STATE_ENUM
{
 OFF,
 ON
};

unsigned int main_state=OFF;
unsigned long change_time; // time of last state change

void Blink_state_machine() {
 // insert your state machine code here
}

void setup() {
 // setup your pins
 pinMode(LED_pin, OUTPUT);
  // setup the first state (OFF)
 digitalWrite(LED_pin, LOW);
 change_time = millis();
  // setup the serial port
 Serial.begin(9600);
}

void loop() {
 // Run the state machine
 Blink_state_machine();
  // You can run other state machines in parallel

 // Read the value from the LDR and print to serial port
 Serial.println(analogRead(A0));
}
```

Two different state machine implementations are presented below, using blocking and non-blocking code.

Mount the required circuit (an LED and an LDR) and test the given code with both possible solutions. Discuss the differences with your professor. Which solution do you think is best? Why?

TIP: DO NOT remove the circuit you used in 3.1 and 3.2. **You will need it later**. Instead, add an LDR to the circuit you used for the previous example and include a new LED, connected to pin 4.

```
// State machine with blocking code
void Blink_state_machine() {
 switch (main_state){
   case OFF:
     delay(3000);
     // change LED state
     digitalWrite(LED_pin, HIGH);
     main_state = ON;
     break;

   case ON:
     delay(3000);
     // change LED state
     digitalWrite(LED_pin, LOW);
     main_state = OFF;
     break;

   default:
     // should never get here
     main_state = OFF;

     break;
 }
}
```

```
// State machine with non-blocking code
void Blink_state_machine() {
 switch (main_state){
   case OFF:
     // when in OFF state, do nothing
     // (just wait to change)
     if((millis() - change_time) >= 3000) {
       change_time = millis();
       // change LED on the transition
       digitalWrite(LED_pin, HIGH);
       main_state = ON;
     }
     break;

   case ON:
     // when in ON state, do nothing
     // (just wait to change)
     if((millis() - change_time) >= 3000) {
       change_time = millis();
       // change LED on the transition
       digitalWrite(LED_pin, LOW);
       main_state = OFF;
     }
     break;

   default:
     // should never get here
     main_state = OFF;

     break;
 }
}
```

3.4 Parallel state machine execution

One of the advantages of using state machines is that you can design independent parts of the system as independent state machines, that are then executed in “parallel”.

Create a system that combines the functionality implemented in 3.2 and 3.3 (with non-blocking code). For this, you should run both state machines in the main loop.

TIP: you can include the complete state machine of 3.2 in a function, as done in the example of 3.3, and call both state machine functions from the loop.

4. Nested state machines

In the previous examples, you have used state machines to toggle between static states, in which the system does not change during each state. However, a program may need to have a dynamic behavior in a specific state.

Consider the example 3.1 (or 3.2). Instead of the GREEN state, you should now include a CYCLE state, which lights each of the LEDs for 200ms, cyclically, in order (RED→YELLOW→GREEN→RED…), for as long as the system is in this state.

In this case, the changing state has a dynamic behavior, which also needs to be controlled. For this reason, it might be helpful to create a different state machine to control the execution of this specific state. The inclusion of a state machine in a specific state of another state machine is frequently denoted as a nested state machine. This is extremely helpful when creating complex programs, since it separates the complexity of each specific state.

Implement the functionality described above. Remember that the user must be able to switch mode at any moment, so using delay() to control the timing might not be a good idea.

Tip 1: You can use millis() to control the timing of the LED changes without blocking the code, as shown before.
Tip 2: You can use a second state machine to implement the CYCLE state (with states RED, YELLOW and GREEN). This state machine can be developed in an independent function and be called only when the main state machine is in the CYCLE state.

5. Interactive Traffic Lights

Observe the circuit in **Figure 4**. Implement this circuit and write a program for your Arduino UNO to control the traffic lights for cars and pedestrians. Notice that the circuit has a push button.

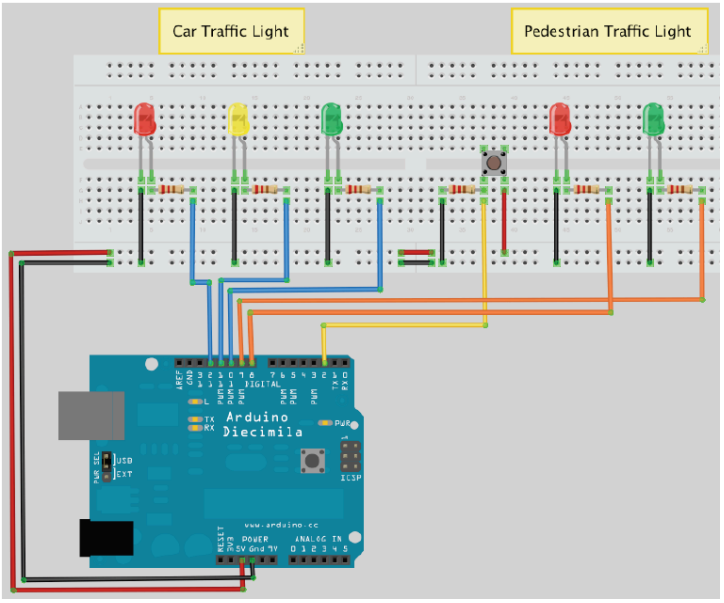

Figure 4. Interactive traffic lights.

The operating cycle should be as follows:

- By default, car lights remain permanently green and pedestrian lights red.
- When the user presses the push button, the following sequential steps should occur:
  i. Car lights should change to yellow during 1 second, and then to red.
  ii. After 2 seconds, the pedestrian lights change to green and stay green for 3 seconds.
  iii. Then, the green light for pedestrians should start blinking for 3 seconds.
  iv. Then, the pedestrian lights change to red.
  v. Finally, after 2 seconds car lights change back to green.

Remember that additional button pushes (after the lights have started to change for pedestrians) should be ignored. Upload and test your program.

If you implement the program using a state machine (recommended), draw the state machine first and show it to your professor.

6. Day and night traffic lights (Bonus)

Include an LDR in your circuit and extend the previous work so that:

- At night, the traffic lights remain in the default mode (green for cars and red for pedestrian) indefinitely, exactly as described before.
- During the day, car lights are only green for a maximum of 5 seconds, and then automatically commute to pedestrian mode, even if the button is not pressed.

Upload and then test the circuit by covering and uncovering the LDR.