1 Objectives

By the end of this lab, you will:

- Create a matrix circuit of inputs and outputs
- Read input and output information using this circuit
- Develop a better understanding of the circuit and I/O operations necessary for the Nim project

Materials: We will provide the parts necessary for this lab (except for the Arduino and breadboard) if you do not already have them.

To complete the lab, you will need the following:

- 3 LEDs (RGB or single-color)
- 1 switches
- Resistors (1kΩ and 10kΩ)

2 Pre-lab: Pull-up resistors

In the last lab, we connected a push-button switch to your Arduino using a pull-up resistor, which gave the switch’s input pin a “default state” when unpressed by using an extra resistor.

Since connecting switches and other inputs is a common practice, most Arduinos (and other microcontrollers) contain configurable, built-in resistors on their inputs to provide this extra resistor without any additional wiring. Before starting this lab, we will test out this functionality to ensure it is supported on your Arduino, and show you how to use it.

On the Arduino Uno, the built-in resistor for this purpose is configured as a pull-up resistor. Logically, a pull-up resistor connected with a switch looks like the schematic in Figure 1.

However, the wiring for this circuit is much simpler than in the previous lab. Connect a switch to your Arduino as shown in Figure 2.

Once connected, make a new sketch in the Arduino IDE and configure the pin using pinMode(number, INPUT_PULLUP); instead of the standard configuration value of INPUT. This will configure the hardware to internally connect the pull-up resistor as shown in Figure 1.
In the \texttt{loop()} function of your sketch, read from the switch to examine its value (either print out its value in the serial console, or use it to control an LED). You should notice that the logic for the switch is \texttt{inverted} from what we usually expect: the pin returns logic high when the switch is not pressed, and logic low when the switch is pressed. This is due to the behavior of the pull-up resistor: when the switch is open, pull-up resistor connects the input to 5V, which causes it to read as high by default. Keep this in mind as you build the circuits in this lab and for your future projects.

3 Creating matrix of LEDs

This task will demonstrate a starting point for creating a matrix of LEDs. Using your breadboard, create the circuit shown in the schematic in Figure 3.

This circuit represents a single row of the matrix. You will notice that all of the connections for this circuit are to output pins. In this schematic, the three pins connected to the anodes of the LEDs control the “columns” of the matrix, while the digital pin connected to the anodes of all three LEDs controls a single row.

In software, the procedure for setting the state of each LED is as follows:
• Select this row of LEDs by assigning a LOW signal to its row select pin
• Set the state of each LED (HIGH or LOW) in this row using the column select pins
• Wait some period of time (50ms should do)
• Deactivate this row of LEDs by assigning a HIGH signal to its row select pin.
• Repeat this process for any additional rows (in this lab, we do not have any)

Here, a row is “activated” by driving it low. You can visualize this on the schematic: by placing a logic 0 (0V) on the row pin, current can flow from one of the column pins to the row select pin, turning on the LED.

In order to wait some period of time in step 3, you should not use the `delay()` function. While it may work in a simple scenario like this lab (you can try it to start), it is not a good practice since it prevents the Arduino from performing any additional computational steps. Instead, use the `millis()` function to read the current timestamp and use comparisons.

**Task**: make an array of booleans that defines the state for your LEDs. Configure your sketch such that pressing the button cycles through all LED states.

**Note**: you may notice that pressing the button skips some LED states. You can add a delay to handle this for now—we will resolve it in the future labs.

## 4 Debouncing

Switches have some fun physical properties. When pressing a switch, there is a small window of time (on the order of milliseconds) when the switch may *bounce* between high and low states before settling to a consistent state, as shown in Figure 4. This is caused by the metal contacts in the switch literally bouncing after being pressed.

If your software performs an action for each switch state change without accounting for the bounce, the output may be unpredictable. We can account for this hardware issue using software by reading the switches using an algorithm like the following:

- If the switch state has changed, store its current value into a temporary variable
Wait a short period of time (e.g. 15 milliseconds)

If the current switch state is the same as the value in the temporary variable, perform the state change action.

**Task:** Implement this debouncing step in your code (again, without using delay()). Keep in mind the time spent while the LEDs are activated and how it might affect or help your code. Also, remember that this algorithm is only a template: depending on how you want to register button events (separate press/release events, duration, etc.), you can use your own debouncing routines to suite your needs in your own circuits.

Note that debouncing is not always necessary. For example, if a switch state changes only once in the life of a program, then an action can be taken immediately on the switch state change because no further changes will have an effect.