CSIS10C Lab 2b

Arduino Uno,
LEDs, Pushbuttons,
and Finite State Machines

Agenda

Part A

1. Serial I/O and LEDs
2. Debugging with Serial I/O
3. Setting up a State Machine based on Serial I/O

Part B

1. Programming a Finite State Machine
2. Reading a Pushbutton
    - Using pushbuttons to change flashing rate
    - Counting button presses
3. Challenge -- Programming Arduino Finite State Machine

Concepts
Finite State Machines
Serial Input/Output
Digital Output: LEDs
Digital Input: Pushbuttons
Techniques
Debugging
Code blocks
Incremental changes
Editing

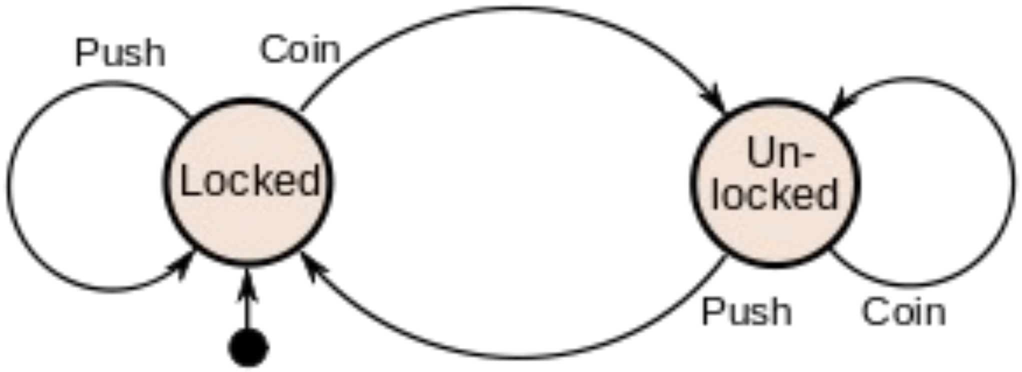
Lab 2B Part 1

- In this first part of Lab 2B, we will
  - Learn about finite state machines
  - Learn how to use the Serial Monitor
  - Determine how to use LEDs in your projects
  - Use the serial port to debug our program
  - Setup a finite state machine

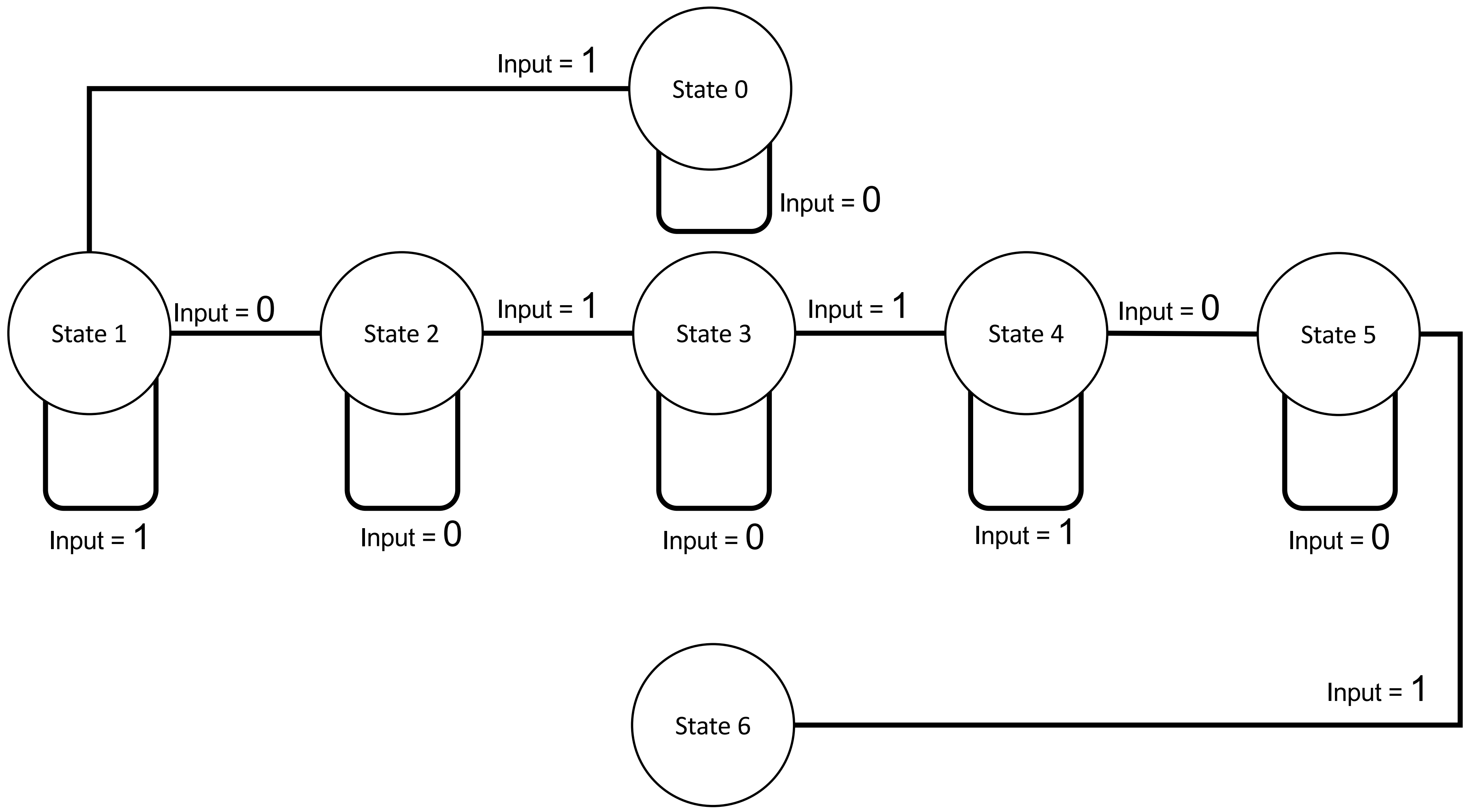
Finite State Machines

- A deep topic in computer architectures, robotics
- You can use a special variable (called a “state” variable)
  - Keep track of which state your program is in
  - Each state offers different input / output behavior
  - Pressing a button changes the state of the machine.
  - See https://en.wikipedia.org/wiki/Finite-state_machine

- Example



Example -- Combination lock FSM



Success – Unlocked!

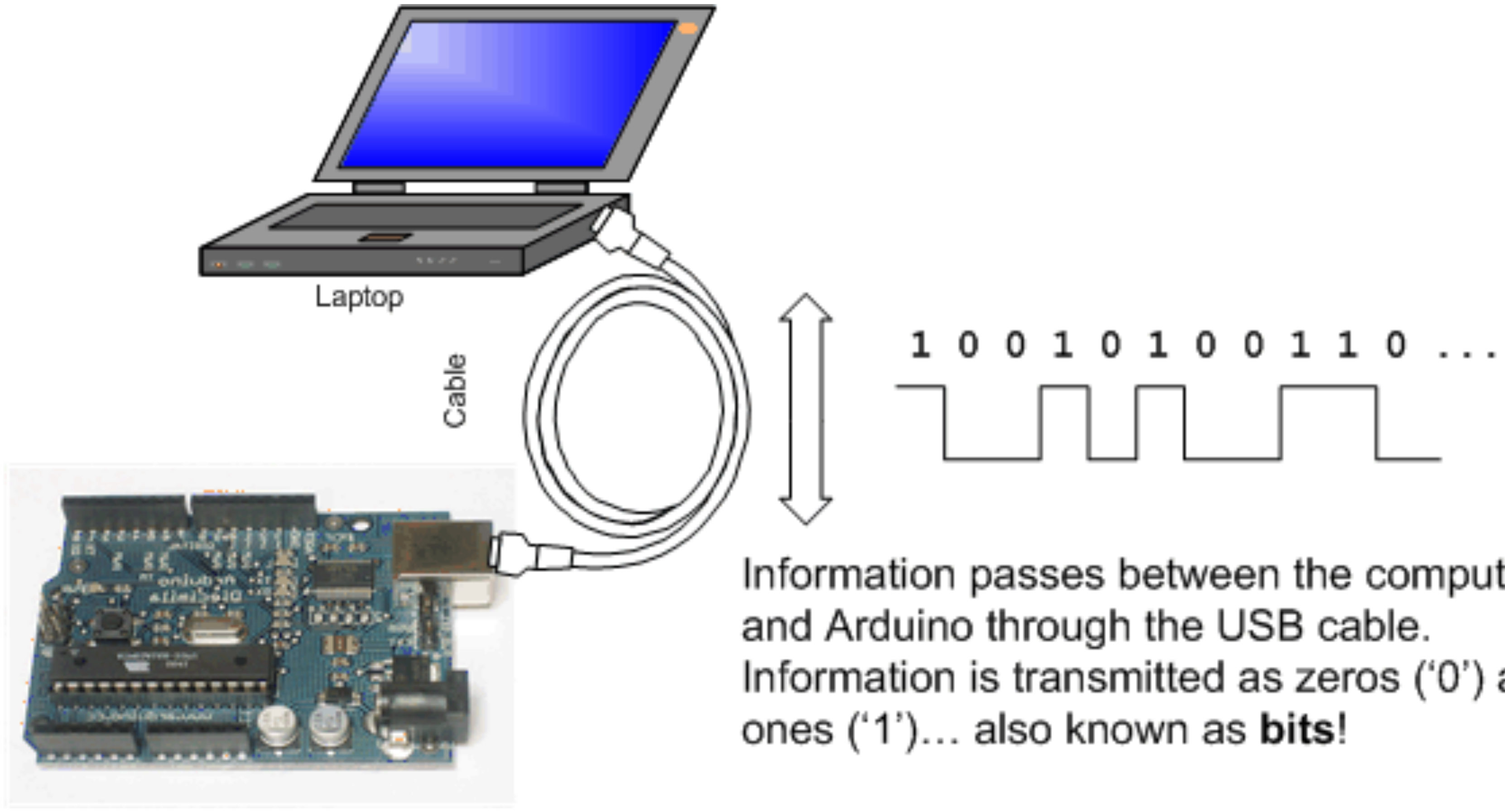
Example Coding a FSM: Psuedocode

```
Get input
If state 0
     if input = 0
          perform  state 0, input 0 activities
     if input = 1
           perform  state 0, input 1 activities
     clear input
If state 1
     if input = 0
          perform  state 1, input 0 activities
     if input = 1
           perform  state 1, input 1 activities
     clear input...
```

Introduction to Serial I/O

- In computing **input/output** or **I/O** (or informally **IO**) is communication between a computer and other devices
- Serial I/O is how we get Arduino to communicate with our PC
- All Arduino boards have at least one *serial port*
- A serial port is a connector by which a device that sends data one bit at a time may be connected to a computer
- One way to transmit data through the serial port is with a USB cable
- Like data is stored in a computer using binary numbers, data is transmitted in binary

Serial Data Transmission between an Arduino and PC

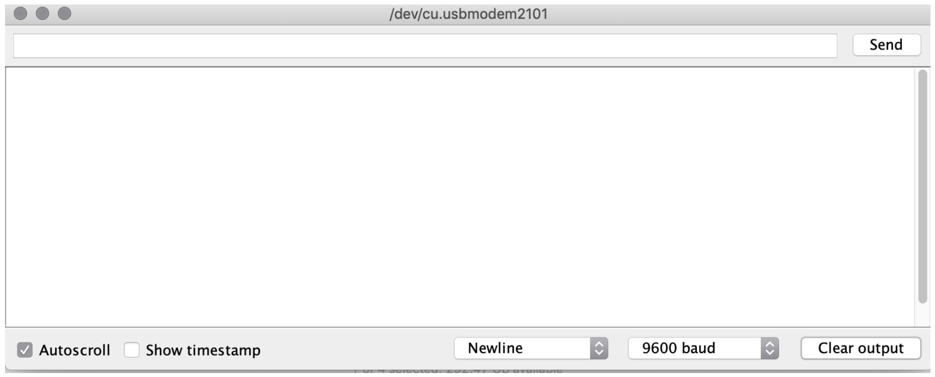


Information passes between the computer and Arduino through the USB cable. Information is transmitted as zeros ('0') and ones ('1')… also known as **bits**!

Serial Monitor

- A serial program allows our computer to display messages sent by the Arduino board
- Windows: *PuTTY, Arduino Serial*
- Linux: *minicom, serial, Arduino Serial*
- macOS: *minicom, Serial, Arduino Serial*

Arduino Serial



/dev/cu.usbmodem2101

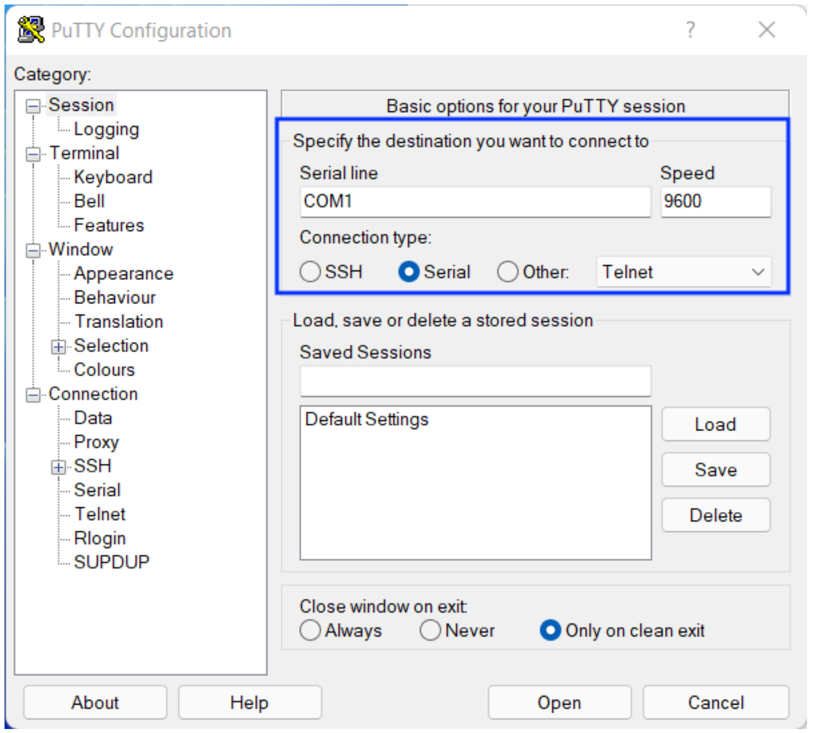
Serial Monitor Setup

The basis of communication with the Arduino Uno is:

- 9600 baud
- 8 bits, no parity, 1 stop bit
- Specify serial port…COM3, /dev/tty… etc

Here are examples of how to set common serial programs

- minicom -b 9600 -D /dev/cu.usbmodem3101
- Arduino Serial 9600 baud, No newline



Basic options for your PuTTY session

Specify the destination you want to connect to

Serial line: COM1

Speed: 9600

Connection type: SSH [✓ Serial] Other: Telnet

Load, save or delete a stored session

Saved Sessions

Default Settings

Close window on exit: Always Never [✓ Only on clean exit]

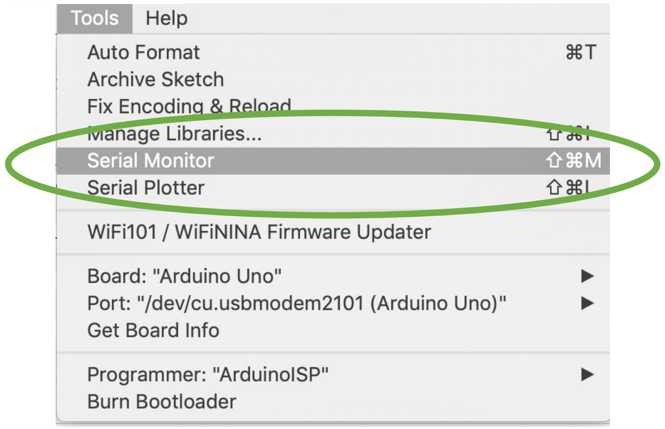
Serial I/O Setup

- To get started, we open the serial monitor by:
  - pressing the special button or using the menus on the Arduino

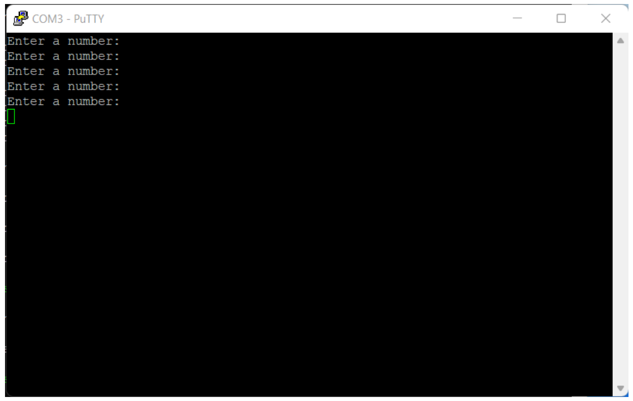
  Or
  - Running minicom in a terminal window, macOS and Linux

  Or
  - Running PuTTY or Serial in Windows or macOS

Arduino IDE



Windows PuTTY



```
Enter a number:
Enter a number:
Enter a number:
Enter a number:
Enter a number:
```

AVR-libC: The C Library for the AVR family

- The Arduino Uno uses an AVR ATmega328P microcontroller
- The AVR Libc package provides a subset of the standard C library for AVR microcontrollers. In addition, the library provides the basic startup code needed by most applications.
- http://www.nongnu.org/avr-libc/
- We'll incorporate AVR Libc in functions starting with serial I/O
- This allows us to use Standard C functions such as puts(), getchar() and fprint() instead of the Arduino class Serial.print()

AVR Libc: stdio.h standard IO faculties

http://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html

- To use the Standard C functions, you must have the following 3 elements (in **bold**) in your code:

(*near the top of the file*)

#include <stdio.h>

- Provides the Standard C I/O functions

#include "uart.h”

- Identifies the UART polling routines as streams, stdin and stdout

(*in main()*)

init_serial;

- This sets up the code to use the Uno serial hardware in a polling fashion (slow, 9600 baud)
- init_serial is a macro so () are not used at the end

- Review *serialio* in the examples folder as a simple implementation

Standard C I/O functions

http://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html

- **getchar()** - function reads a character from stdin. It returns the character, or EOF in case end-of-file was encountered or an error occurred.
  - EX: input = getchar();
- **printf(**string, variables**)** - outputs values to stdout under control of a format string passed in fmt (see vprintf for all formatting options)
  - EX: printf(“This is your input %d\n”, input);
- **puts(**string**)** - write the string to stdout.
  - EX: puts("Serial I/O Test");

Example Code: serialio

```
#include <stdio.h>
#include "uart.h"
int main(void) {
     init_serial;
     char input;
     puts("Enter a number");
     while((input = getchar())!= EOF) {
          printf("You entered %c\n", input);
    }
    return 0;
}
```

Adapted from K&R The C Programming Language V3 page 17

First Exercise: 1_serial

- **Objective(s):**
  - Demonstrate the ability to compile/link/load a program on the Uno
  - Demonstrate the ability to use a serial monitor with the Uno
- **Steps**

1. Go to folder 2B and open folder 1_serial
2. The program main.c is already complete and ready to use
3. Compile and upload it to the Arduino (*make flash*)
4. Open your serial program (Arduino Serial Monitor, PuTTY, minicom)
5. Enter numbers and confirm they are printed back out

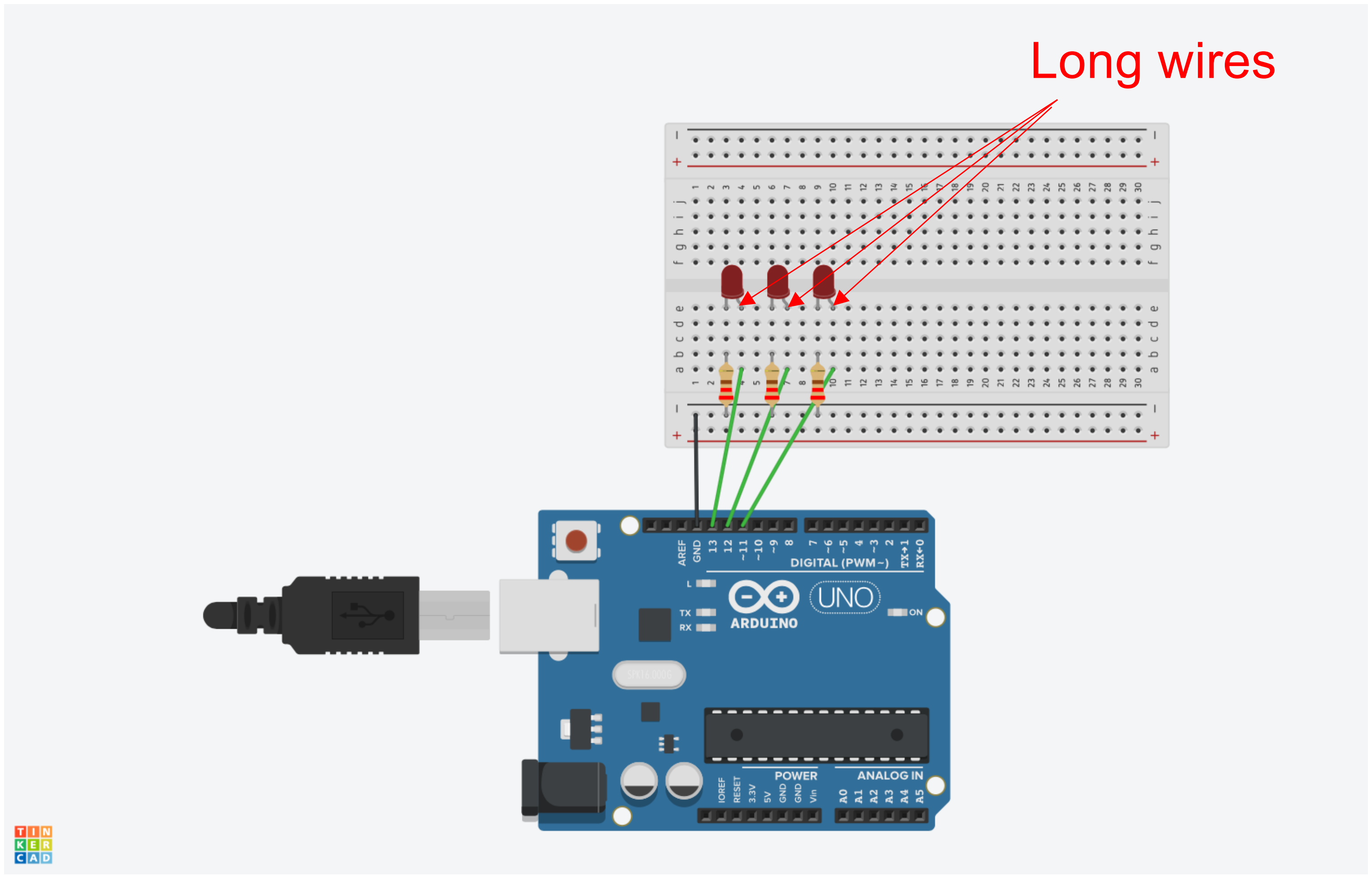
Technique: Debugging using the Serial Port

- Many times, we’re not sure of why a program isn’t working correctly
- One way to debug the program is to insert print statements which show where the program is and possible variable values
- This along with hand walking your code allows you to determine where the program is failing
- Debugging with print statements is the first step to good debugging technique

Lab 2B – Exercise 2

Wire up the Arduino with 3 LEDs

(When you finish today's lab, DON'T dismantle your circuit!)



If you have issues, review https://sites.google.com/a/cabrillo.edu/cs-11m/schedule/memory Exercise 1

Technique: Code Blocks

- Develop your code in blocks that have specific tasks (turn on a set of LEDs, input a number via the serial port etc)
- Title the block with a comment, so you know exactly what it does:

```
// Display binary 7 on 3 LEDs
```

- Then copy and paste the blocks as appropriate
- You might need to update certain values, such as changing a test from a 0 to a 1 or change the display of a number from 3 to a 4
- We will begin to turn code blocks into functions in the next few weeks
- In the meantime, there will be a lot of code repetition, however if the block is properly debugged prior to copying, it will reduce the amount of work writing your code

Numbers in C

- Using the Serial monitor and getchar(), provides a number in ASCII format and it will need to be changed to a decimal number.
- ASCII is a standard format for text. Notice that ASCII numbers are sequential. https://www.ascii-code.com
- To convert an ASCII number to decimal:

```
int number = getchar() – ‘0’;
```

- Binary number use only 0 and 1 to represent a number
- https://www.exploringbinary.com/decimal-binary-conversion-table/
- For our purposes:

Decimal	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Binary	000	001	010	011	100	101	110	111

Second Exercise: 2_binary

- **Objective(s):**
  - Demonstrate the ability to input data to the Uno
  - Demonstrate the ability to program decision-making
  - Demonstrate the ability to copy/edit code blocks to decrease development time
- **Steps**
  1. Open folder 2_binary
  2. The program main.c is partially complete. The 3 LEDs need to show the correct binary display for each number, 0-7 entered.
  3. Numbers 0 and 1 are correct, following the existing code, and add additional blocks for 2-7.
  4. Compile and upload it to the Arduino (*make flash*)
  5. Open your serial program (Arduino Serial Monitor, PuTTY, minicom)
  6. You should be able to enter a number 0-7 on the keyboard and the binary equivalent will be displayed on the LEDs.

Thoughts on Coding

- Don’t try to simplify the code too soon. Its best to write the code as clear as possible, then if, desired, simplify the code.
- For example: this state machine, there are two decisions:

1. What state?

```
if state == 0 {
   do something }
if state == 1 {
    do something }
```

2. What is the next input?

```
if num == 0 {
    do something }
else if num == 1 {
    do something }
```

3. **Put it together:**

```
if state == 0 {
     if num == 0 {
      do something }
     if num == 1 {
      do something }
}
if state == 1 {…

If state == 2 {…
```

Third Exercise: 3_serial_state

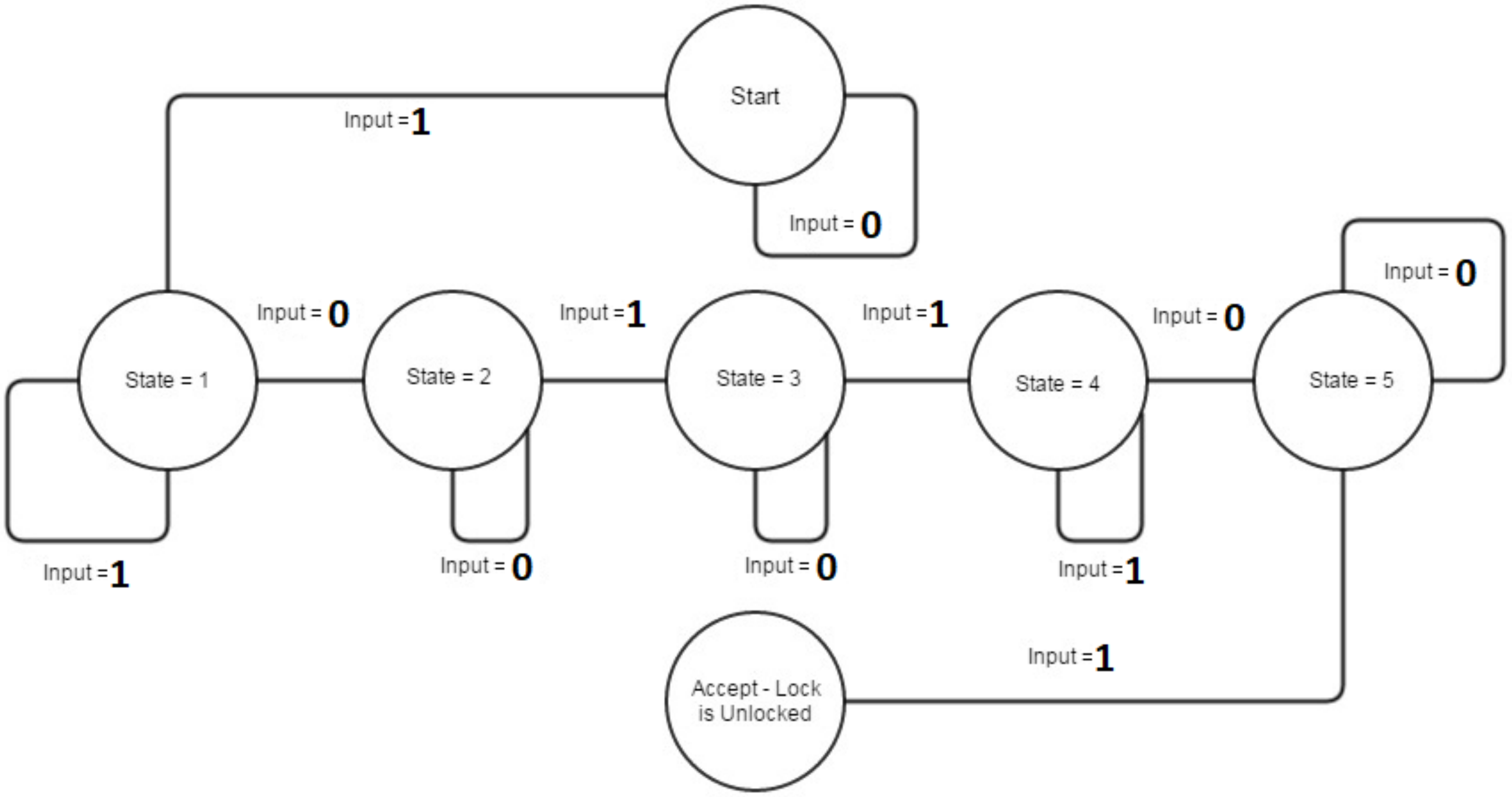
Make a Finite State Machine

- **Objective(s):**
  - Demonstrate the ability to program a finite state machine
- **Steps**

1. Open folder 3_serial_state
2. Use the pseudo-code FSM example to create a C version of the FSM lock
3. The program main.c is partially complete. It contains sufficient code to help you understand how to convert the pseudocode into C.
4. Compile and upload it to the Arduino (*make flash*)
5. Open your serial program (Arduino Serial Monitor, PuTTY, minicom)
6. You should be able to enter a number 0 or 1 and move through the finite state machine.

Exercise 3 – continued…

State Definitions

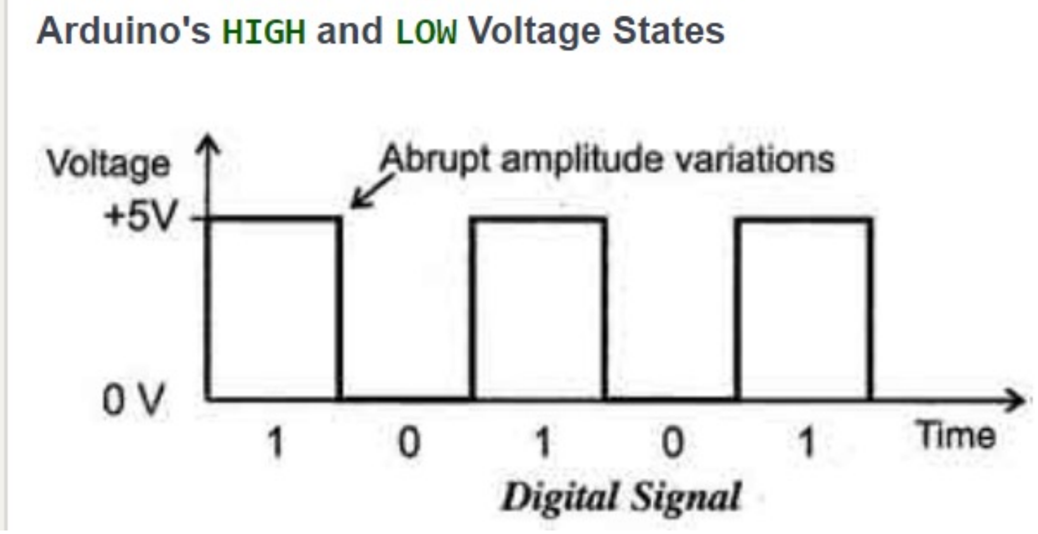


Now test your state machine – verify the state changes correctly based on the console input. Note that you will need to change your code in the "if" statements to match the inputs above.

Lab 2B Part 2

- In this second part of Lab 2B we will
  - Experiment with pushbutton switches
  - Add switches to change the state of our state machine
  - Notice how pushbutton presses are sometimes ignored
  - Improve performance using hardware interrupts

Adding Pushbuttons to your project – Arduino Digital I/O


Arduino's HIGH and LOW Voltage States
Voltage
+5V
0 V
Abrupt amplitude variations
1
0
1
0
1
Time
Digital Signal

Digital I/O

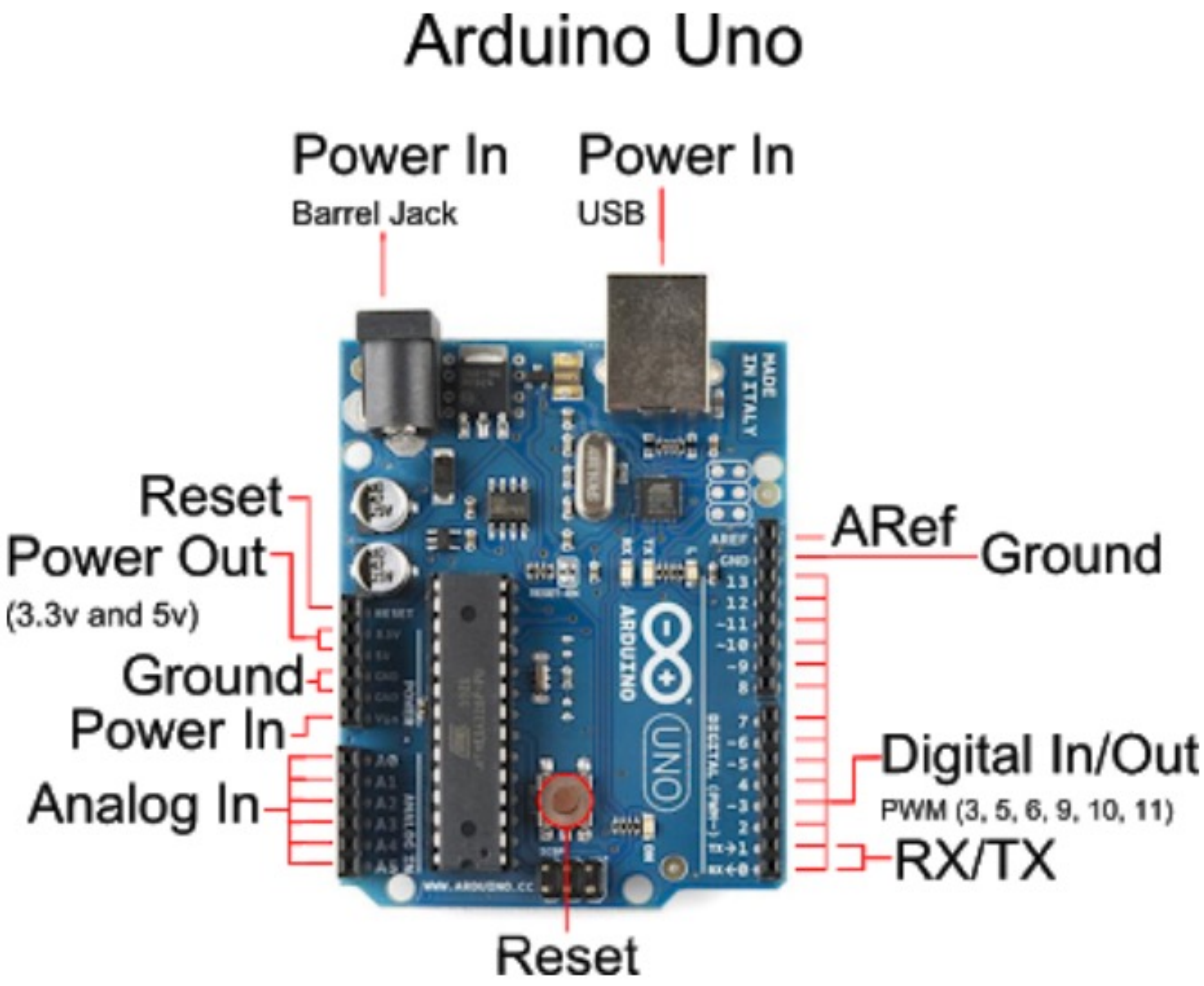
- Arduino is intended for physical computing and thus has many I/O possibilities
- Much of the I/O is through attaching electronics to the Arduino pins
- These attachments can be either digital (on or off) or analog with various voltage levels
- We will discuss Digital I/O today

Representing Two States

- Digital means there are two states
- In electronics, we represent these two states as low and high voltages
- Recall that voltage is the difference in electric charge between two points
- Typically, the Arduino uses 0 volts (ground) as low and +5 volts as high
- Arduino designates these states as `LOW` and `HIGH`
- The LOW and HIGH states can be input and output using Arduino's digital pins

Digital Pins and I/O Functions

- The Arduino has many digital pins available
- For example, the Arduino Uno from our kit has 14 digital pins numbered 0 to 13
- These pins can be used as either input or output using the functions described below
- Some of the pins have specialized functions in addition
- For example, pins 0 and 1 receive (RX) and transmit (TX) serial data



Digital I/O Functions

Name	Description
pinMode()	Calling `pinMode(pin, mode)` configures the specified *pin* to behave either as an `INPUT`, `INPUT_PULLUP` or `OUTPUT`.
digitalWrite()	Calling `digitalWrite(pin, value)` writes a `HIGH` or a `LOW` value to the specified digital pin.
digitalRead()	Calling `digitalRead(pin)` returns a `HIGH` or a `LOW` value from the specified digital pin.

Coding Digital I/O

- We can write Arduino commands to control the digital pins
- To set the pins to input or output, we call the `pinMode(pin, mode)` function
- Where:
    - *pin*: the digital pin to set (0 - 13)
    - *mode*: one of `INPUT`, `INPUT_PULLUP` or `OUTPUT`
- We will look at basic examples of input and output in this section

Digital Output

- We have made use of an LED attached to digital pins for output, such as blinking with pin 13
- With digital output set to a HIGH state, the pin provides up to 40 milliamps (mA) of current at 5 volts
- The pins are useful for powering LEDs which typically use less than 40 mA
- Loads greater than 40 mA, like motors, require a transistor or other interface circuitry
- Pins configured as outputs can be damaged or destroyed if they are connected to either the ground or positive power rails
- We must protect the output using component such as a resistor
- Digital output is coded using the `digitalWrite(pin, value)` function
- Where:
    - *pin*: the digital pin to set (0 - 13)
    - *value*: either `HIGH` or `LOW`
- For example, from the blink sketch:

```
void loop() {
  digitalWrite(13, HIGH);
  delay(1000);
  digitalWrite(13, LOW);
  delay(1000);
}
```

Digital Input

- When set to INPUT the digital pins can read a voltage as HIGH or LOW
- HIGH is when a voltage greater than 3 volts is present at the pin (5V boards)
- LOW is when a voltage at the pin is less than 3 volts
- When set to INPUT, the pin is in a *high impedance* state equivalent to a series resistor of 100 *M*Ω in front of the pin
- Since very little current gets through, the pin has a very low effect on the circuit it is sampling
- Digital input is coded using the `digitalRead(pin)` function
- Where *pin* is the digital pin to read (0 - 13)
- The function returns a value of `HIGH` or `LOW` that we can assign to a variable
- For example:

```
int buttonState = digitalRead(pushButton);
```

- The following code uses `digitalRead()` to measure a pin on an Arduino
- The Arduino is connected to a pushbutton circuit, which we explore in the next section

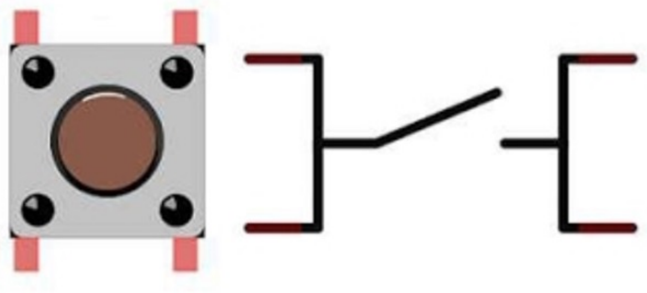
Example Code Calling `digitalRead()`

```
int pushButton = 2;

void setup() {
  Serial.begin(9600);
  pinMode(pushButton, INPUT);
}

void loop() {
  int buttonState = digitalRead(pushButton); // read pin
  Serial.println(buttonState); // display state
  delay(100);
}
```

Digital Input from Pushbuttons

- One use for digital read is to test if a button is pressed
- Buttons, or switches, connect two points in a circuit when pressed
- When not pressed, there is no connection between the two sides of the button
- The following image shows a pushbutton and its schematic symbol

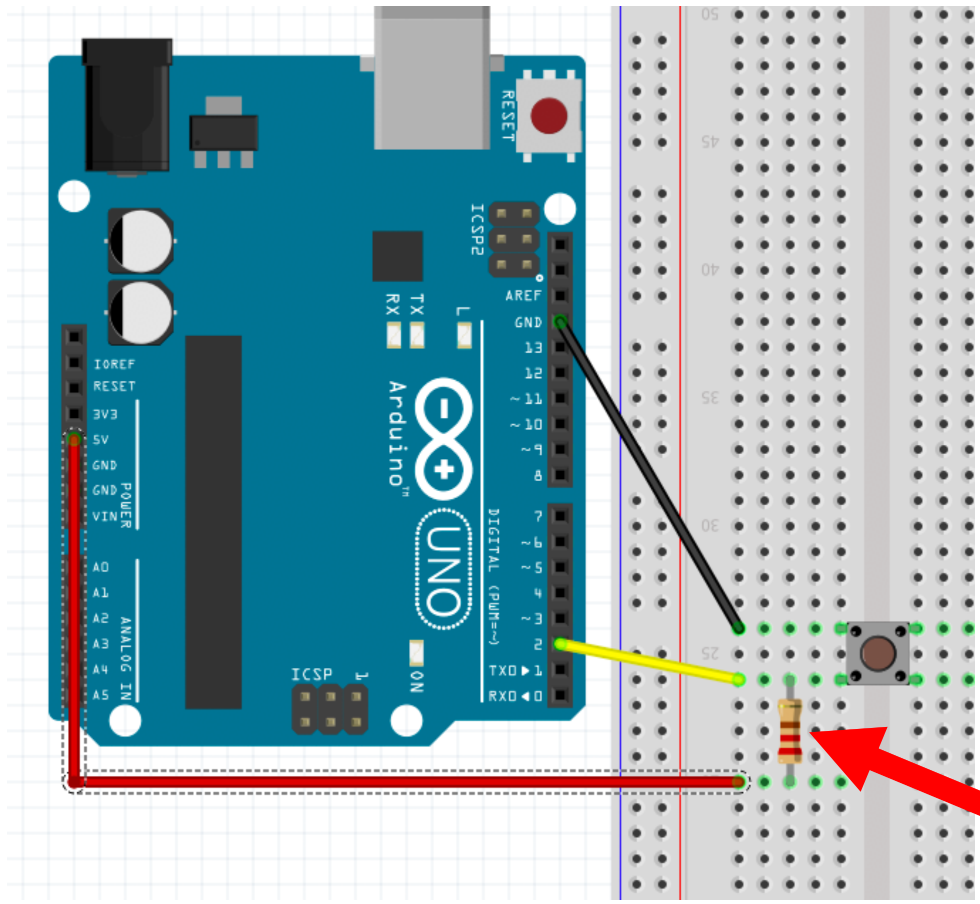
Breadboard Pushbutton and Schematic Symbol



Floating Input and Pull-Up Resistors

- Reading a switch that is open will result in a pin that is "floating"
- Digital input pins are very sensitive and will pick up stray capacitance from nearby sources like breadboards, human fingers and wires
- Any wire connected to an input pin will act like a little antenna and cause the input state to change
- To solve the problem, we use a pull-up or pull-down resistor
- The resistor pulls the pin to a known state when the switch is open
- A 10kΩ resistor is often chosen as a pull-up or pull-down resistor
- A kΩ is electrical resistance equal to one thousand ohms
- So our resistor is equal to 10,000 Ohms

Implement a Digital Read with a Pushbutton



1. Wire up the circuit to the left.
2. One pole of the button is connected to pin 2 then 5V, forcing the pin HIGH by default
3. The other side of the pin is connected to GND.
4. When the button is pressed, the metal contacts close and force the pin to ground.
5. The code to test for button presses is shown on below.

This works but there is a better way that doesn't need a resistor…read on.

Internal Pull-up Resistors

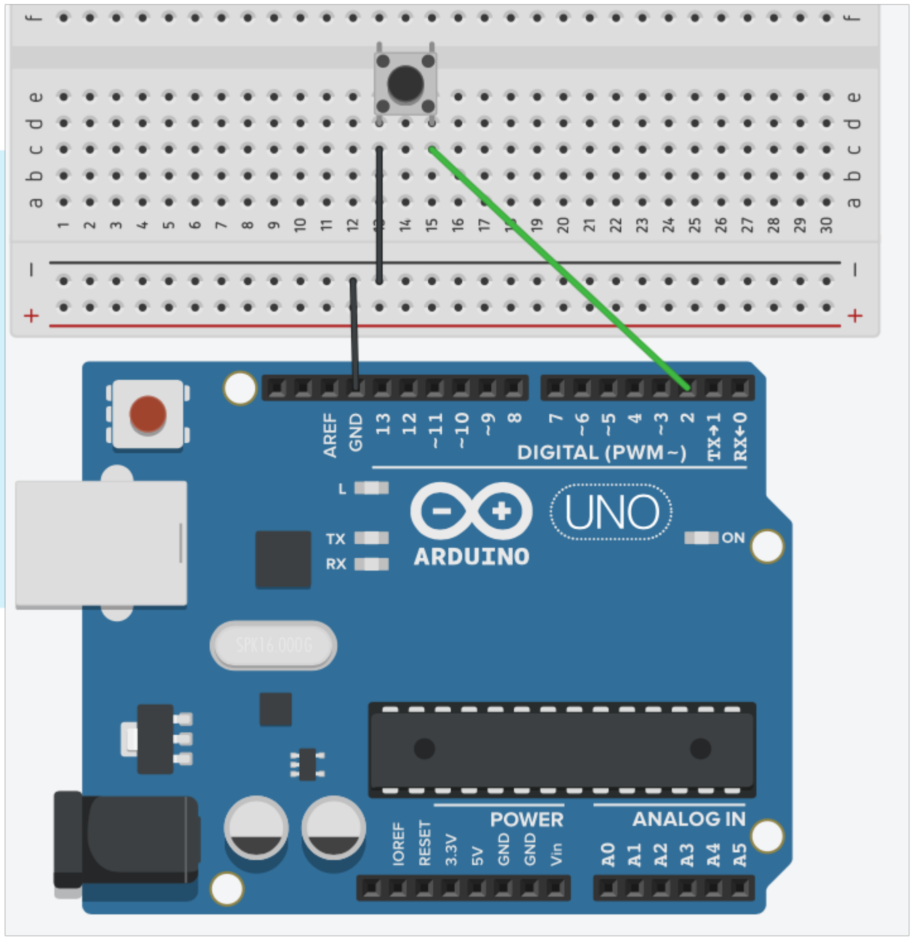
- Arduino has internal pull-up resistors which are controlled by software
- By default the internal pull-up resistors are turned off
- We turn on the pull-up resistors by setting the pin mode to `INPUT_PULLUP`
- For example:

```
pinMode(2, INPUT_PULLUP);
```

- On the Arduino Uno, the value of the pull-up is guaranteed to be between 20kΩ and 50kΩ
- By changing the pin mode we can eliminate the pull-up resistor from our circuit
- The new code and breadboard circuit is shown below

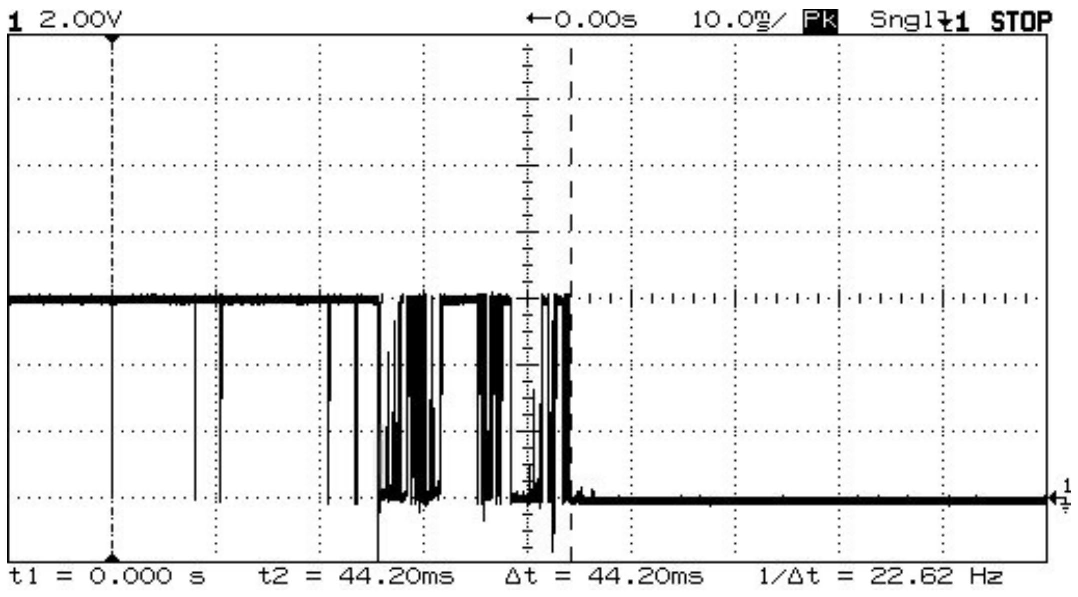
Example Sketch with INPUT_PULLUP

```
int pushButton = 2;

void setup() {
  Serial.begin(9600);
  pinMode(pushButton, INPUT_PULLUP);
}

void loop() {
  int buttonState = digitalRead(pushButton); // read pin
  Serial.println(buttonState); // display state
}
```



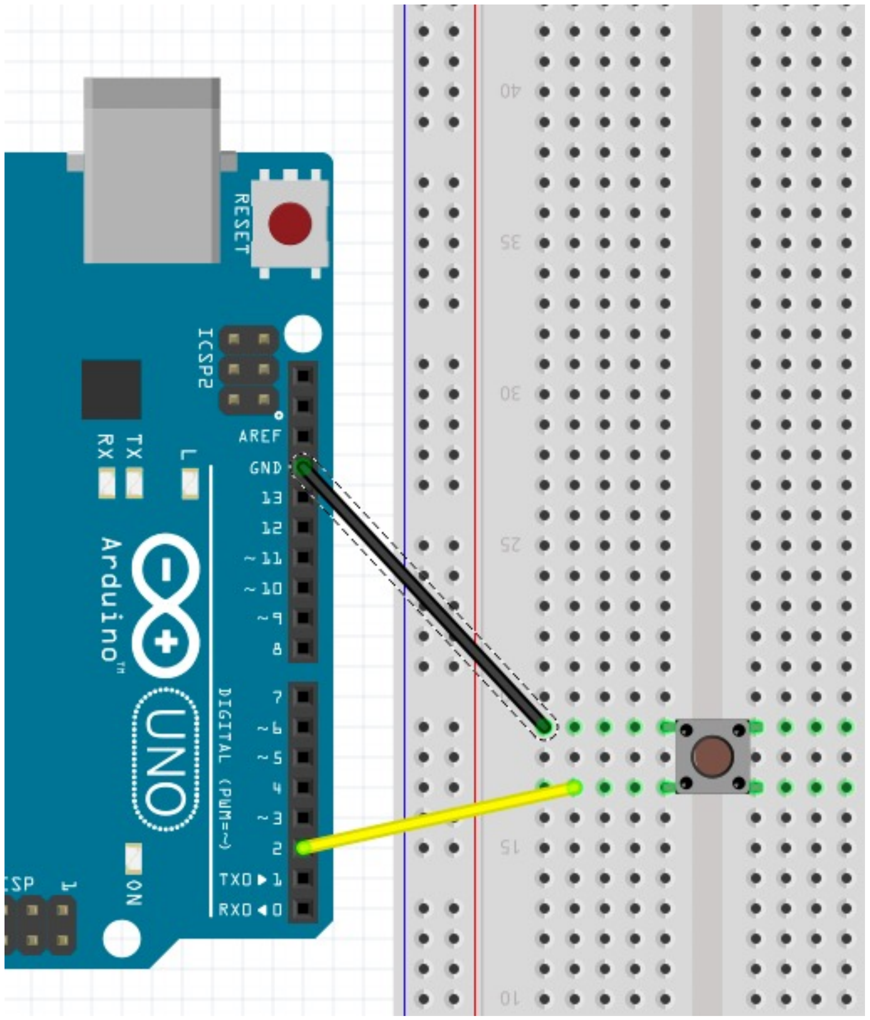
Another issue with switches – “bounce”

- Electrical/mechanical contacts during switching are often erratic
- We can work around this by adding a small delay to our loop function after every switch read
- You may need to wait up to 1/10 or ¼ of a second
- If you have two switches, a single delay after reading both switches should suffice.



Lab 2B – Exercise 4 – Counting Button Presses

Do NOT REMOVE your LEDs



- Start with the Arduino unplugged.
- Make the circuit as shown.

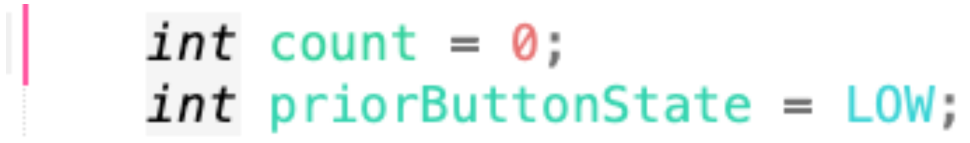
Fourth Exercise: 4_button

- **Objective(s):**
  - Demonstrate the ability to read the state of a button
- **Steps**

1. Open folder 4_button
2. The code is complete. Compile and upload it to the Arduino (*make flash*)
3. Open your preferred serial monitor.
4. What is the value of the button when pressed and not pressed?

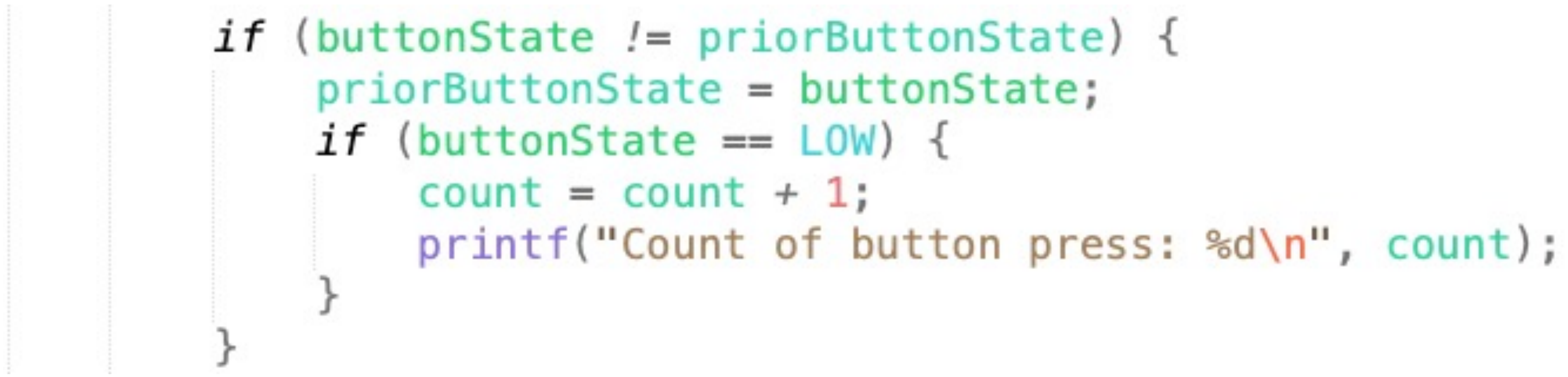
Fourth Exercise: 4_button continued Counting button presses

1. Continue with 4_button from the previous page.
2. Add two new variables after the pinMode line: (*type in*)



```
int count = 0;
int priorButtonState = LOW;
```

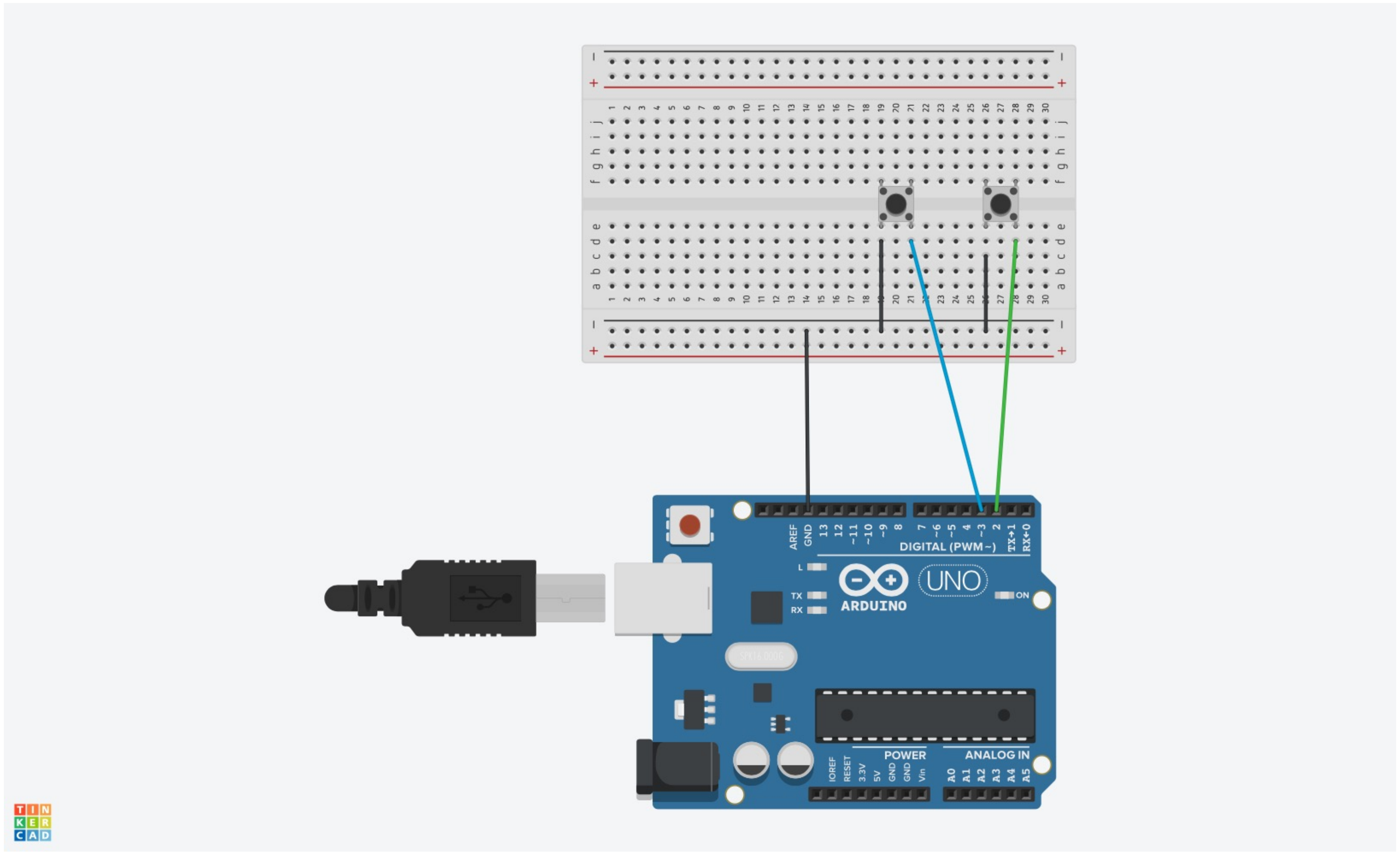
3. Replace the printf("ButtonState…) with the following code:



```
if (buttonState != priorButtonState) {
    priorButtonState = buttonState;
    if (buttonState == LOW) {
        count = count + 1;
        printf("Count of button press: %d\n", count);
    }
}
```

4. Change the delay value from 1000 to 1.
5. Compile and upload your code to verify it works correctly.
6. When you open the serial monitor, you should see a count of every button press.

Two Button Counter



Fifth Exercise: 5_twobutton

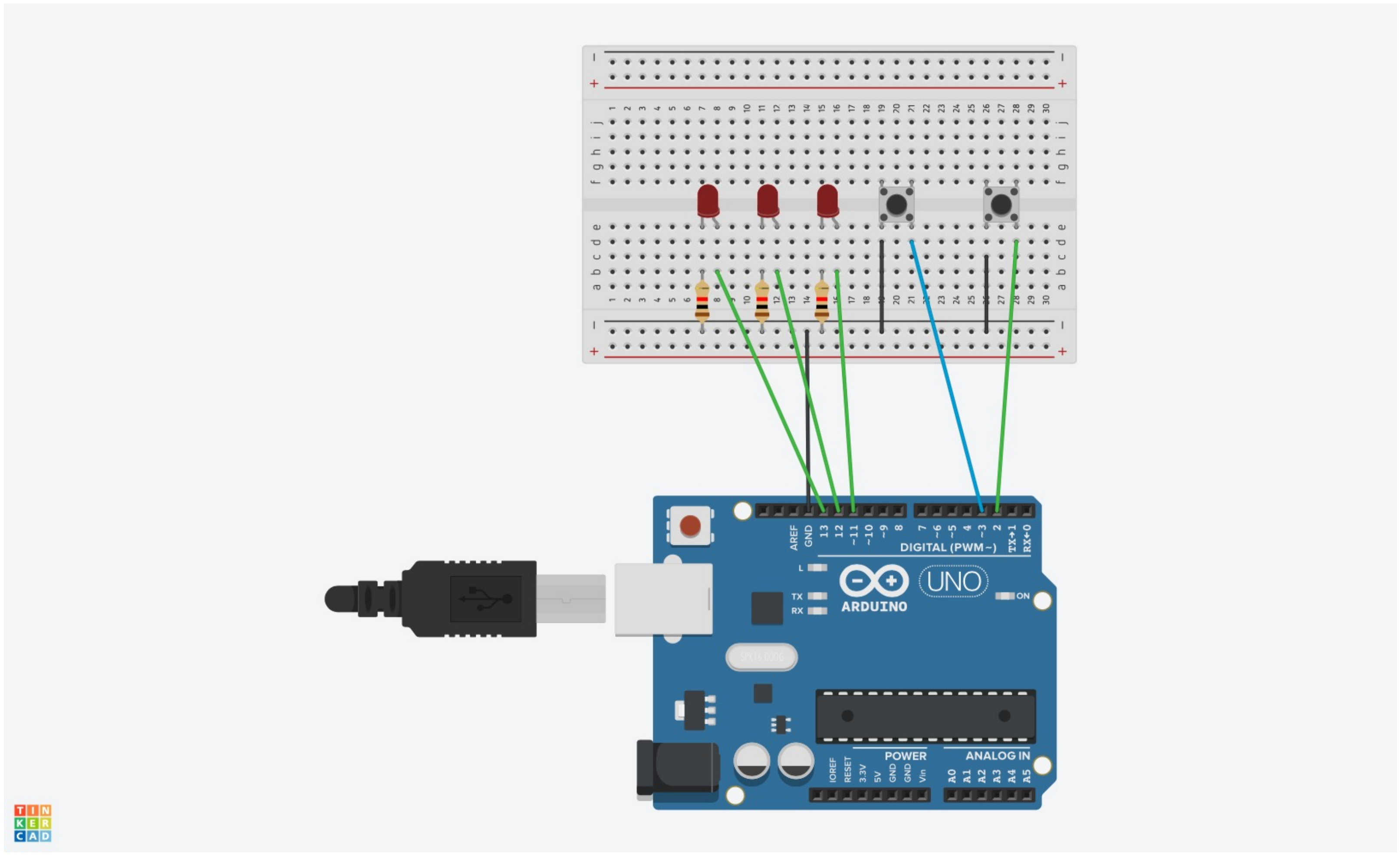
- **Objective(s):**
  - Demonstrate the ability to evolve code
- **Steps**

1. Open folder 4_button main.c, select all of the contents and copy
2. Open folder 5_twobutton main.c. The file is empty, paste the copy contents into the file.
3. Open your preferred serial monitor.
4. What is the value of the button when pressed and not pressed?

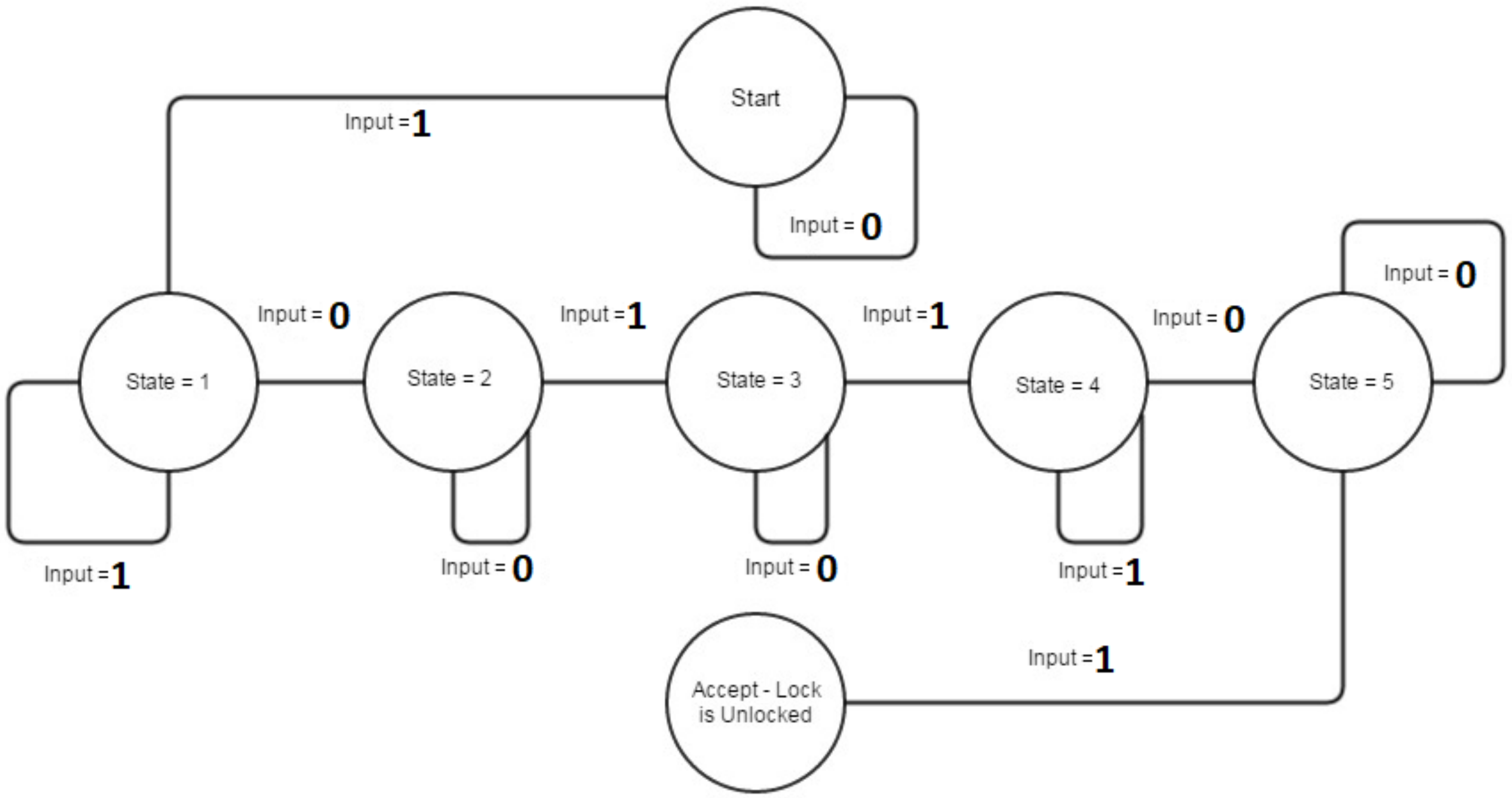
Fifth Exercise: 5_twobutton *continued*
Add a second switch right next to first switch

- Add a second pushbutton connected to pin 3
- Rename the following variables
  - count → count0
  - pushButton → pushButton0   (still equals pin 2 but pressing means input 0)
  - buttonState → buttonState0
  - priorButtonState → priorButtonState0
  - **TEST THESE CHANGES BEFORE ADDING CODE FOR PIN 3!**
- Declare new variables with same name as above but with 1 instead of 0
  - Assign pushButton1 to pin 3
  - Pressing the second pushbutton means input 1
- Add printf() statements so the program prints the count for 0 and 1 presses. If errors, debug!
- Make sure only pressing a button (0 or 1) increases the counter
  - Button releases should be ignored!

Finite State Machine



Assemble the full FSM!



Sixth Exercise: 6_FSM_lock

- **Objective(s):**
  - Demonstrate the ability to evolve code into a new application
- **Steps**

1. Use the programs 2_binary, 3_serial_state and 5_twobutton to create a new program, 6_FSM_lock
2. Think of 3_serial_state as the structure for your finite state machine.
3. Use 2_binary to guide you as to how to light up the LEDs. Have the LEDs indicate the state of the FSM.
4. Use 5_twobutton to guide you as to how to process button presses. Button 0 and 1 are the buttons which change state.
5. Compile, link and load on to the UNO and debug.
6. Are you able to use the sequence **101101** to unlock the lock?

Technique: Hints for full FSM Code

- Remove extraneous code like the counting code to keep debugging simple
- Make incremental changes and test each increment to ensure the program continues to work as expected
- If something doesn’t appear to be working, use printf() to help debug
- Use both Serial Monitor and LEDs while developing code then once working well, remove the Serial commands using only LEDs to show state
- Blink all three LED’s multiple times on Unlock Success

Optional: Need for interrupts

- You may find that your FSM doesn’t seem to respond if you press the button too early or late. Timing is critical because most of the time the FSM is waiting for the LED to change on or off (longDelay or shortDelay).
- You can fix this with a new technique called hardware interrupts
- With this feature you can interrupt the delay statements to allow quick response (interrupt) to button presses (hardware).
- With interrupts, you can also *fine-tune* your software to allow for better debouncing of the button. See this link for more information:

  https://hackaday.com/2015/12/10/embed-with-elliot-debounce-your-noisy-buttons-part-ii/
- Ultimately, by using an interrupt and debouncing, the code for the Finite State Machine can be made much more simple

sysclock and button functionality

- Instead of using the Arduino ISR capability, this framework has new functionality:
  - ***sysclock*** – a Timer/Clock 1 interrupt which happens every 1ms, this period can be adjusted by changing the scalar SCALAR1 in the sysclock.h file. It can be changed to a divide by 8, 64, 256 or 1024, which in turn would provide a period of 8ms, 64ms, 256ms or 1.024secs. For the buttons application, you will want to scan the buttons every 8ms so *SCALAR1 = SCALAR01_8*
  - ***Buttons[i]*** is a struct which uses *buttons[i].uno* to identify which pin on the Uno is attached to a button and *buttons[i].pressed* to determine if the button has been pressed. This allows code that looks like this:

```
    uint8_t i = 0;
       buttons[i].uno = 10;
       pinMode(buttons[i].uno, INPUT_PULLUP);

   /* loop for checking buttons */
    if (buttons[0].pressed) { …code for button 0…}
```

Lab 2B – Exercise 7 – Add interrupt handling to Combo Lock FSM

Additional Notes:

- Add two instances of buttons[] by i=0 and i=1
- Notice from the code in the previous exercise 6_FSM_lock that you may delete all the code which checks the button states, prior button states, and so on. This simplifies your program a lot.
- As the button press check happens every 8ms, this is quite slow for the microcontroller. This creates a *race-condition* in the code, when you wish to check a button being pressed twice in a row. (*This is not the same thing as a debounce.*)
- To resolve this issue, clear the button in the same section of code where you check for the button press, to ensure the press has been cleared.

```
        if (buttons[0].pressed) {
                    /* code when button has been pressed */
                    state = 0;
                    buttons[0].pressed = 0;
                }
```

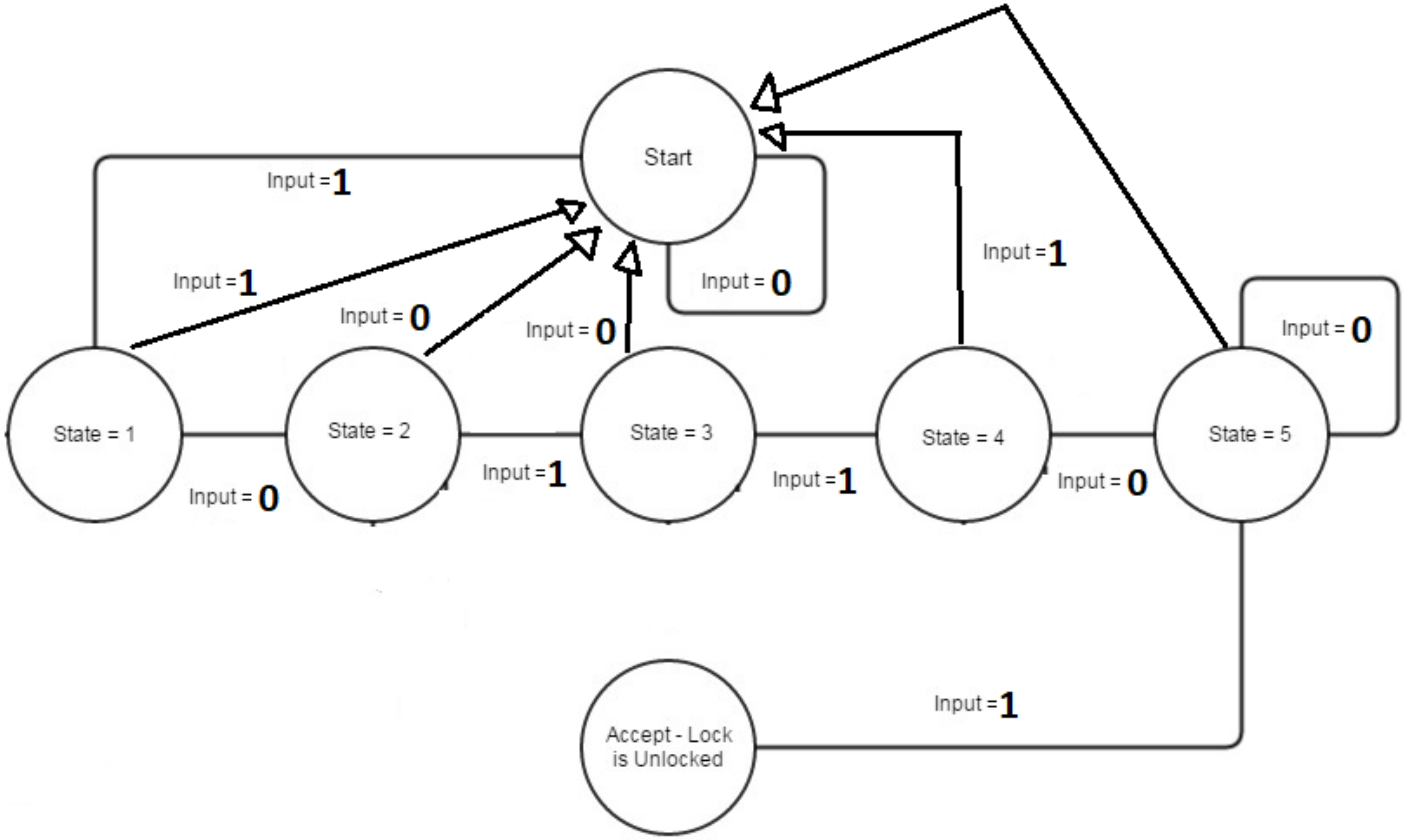
Optional Exercise: 7_FSM_ISR

- **Objective(s):**
  - Demonstrate the ability to simplify the code by adding an ISR
- **Steps**

1. Use the program, 6_FSM_lock
2. Replace the button priorState code with a far more simple:
   if (buttons[0].pressed) { state code }
3. Make sure you resolve the possible *race-condition*
4. Compile, link and load on to the UNO and debug.
5. Are you able to use the sequence **101101** to unlock the lock?

Lab 2B – OPTIONAL

Challenge – Modify your state machine response so pressing the wrong button resets to state 0 (Start)


Start
Input = 1
Input = 1
Input = 0
Input = 0
Input = 0
Input = 1
Input = 0
State = 1
State = 2
State = 3
State = 4
State = 5
Input = 0
Input = 1
Input = 1
Input = 0
Accept - Lock is Unlocked
Input = 1